UNIVERSITY OF CALIFORNIA, SAN DIEGO

Bamboo: Automatic Translation of MPI Source into a Latency-Tolerant Form

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Nhat Tan Nguyen Thanh

Committee in charge:

Professor Scott B. Baden, Chair
Professor Sorin Lerner
Professor David Robert Stegman
Professor Michael Bedford Taylor
Professor John H. Weare

2014

DEDICATION

Dedicated, with the greatest love and respect, to the memory of my grandpa (1934-1994).

EPIGRAPH

Every man’s life ends the same way. It is only the details of how he lived and how he died that distinguish one man from another.

Ernest Hemingway

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor Scott Baden for his guidance and support during the past 5 years. It has been a great honor to be his student. Scott is knowledgeable, approachable, and reliable. His endless love and passion for research and teaching inspired me. I also learned from him how to become a productive, independent investigator. He is terrific, and I simply cannot imagine a better advisor!

I would like to express my gratitude to my dissertation committee members (Sorin Lerner, David R. Stegman, Michael B. Taylor, and John H. Weare) for the advice and direction. I am also grateful to my former committee members (Jason Mars and Lingjia Tang) for their insights and invaluable suggestions. I am in debt to Martha Stacklin, my English instructor at the Center for Teaching Development, for spending her precious time to help me improve my pronounciation skill.

I would like to thank Daniel Quinlan and Chunhua Liao for their time and support during my 2010's summer internship at Lawrence Livermore National Laboratory. I also would like to express my gratitude to Eric Bylaska for mentoring me during my 2011's summer internship at Pacific Northwest National Laboratory.

It has been a great pleasure to work with my wonderful labmates (Pietro Cicotti, Didem Unat, Alden King, Alex Breslow, Natalie Larson, Tatenda M. Chipeperekwa, and Yajaira Gonzalez) and talented visiting scholars from Simula Research Laboratory (Xing Cai, Mohammed Sourouri, Hallgeir Lien, and Johannes Langguth). When running the large-scale systems seminar, I also had the opportunity to work with Allan Snavely and members in his PMaC group. Their thoughtful and constructive comments helped improve the quality of this research.

Last but certainly not least, I would like to express my deepest gratitude to my parents, sister, and relatives for their endless love and support. I am also in debt to my girlfriend and her parents. Thank you everyone for accompanying me on the journey that led to this dissertation.

Chapters 3, 4, 5, and 6, in part, are a reprint of the material as it appears in the article ”Bamboo - Translating MPI applications to a latency-tolerant, data-driven form” by Tan Nguyen, Pietro Cicotti, Eric Bylaska, Dan Quinlan and Scott Baden, which appears in the Proceedings of the 2012 ACM/IEEE conference on Supercomputing.

Chapter 8, in part, uses the material appeared in the Third International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC’13) with the title ”Bamboo - Preliminary scaling results on multiple hybrid nodes of Knights Corner and Sandy Bridge processors” by Tan Nguyen and Scott Baden.

Chapters 5, 6, 7, and 8, in part, are currently being prepared for submission for publication with Scott Baden. The dissertation author is the primary investigator and author of this material.

This research was supported in part by the Vietnam Education Foundation (VEF). I am grateful to the VEF staffs for their great help during the past 5 years.

This research was supported by the Advanced Scientific Computing Research, the U.S. Department of Energy, Office of Science, contracts No. DE-ER08-191010356-46564-95715 and DE- FC02-12ER26118, and in part by NSF contract ACI0326013.

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, and Teragrid resources provided by NCSA, ORNL and SDSC.

Support for computational resources housed in the Computer Science and Engineering Department at UCSD were supported by the National Science Foundation DMS/MRI Award 0821816.

VITA

2009	Bachelor of Engineering, Ho Chi Minh city University of Technology, Ho Chi Minh city.
2014	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

Tan Nguyen, Pietro Cicotti, Eric Bylaska, Dan Quinlan and Scott B. Baden, ”Bamboo - Translating MPI applications to a latency-tolerant, data-driven form”, Proceedings of the 2012 ACM/IEEE conference on Supercomputing (SC'12), Salt Lake City, UT, Nov. 10 -16, 2012.

Tan Nguyen and Scott Baden, ”Bamboo - Preliminary scaling results on multiple hybrid nodes of Knights Corner and Sandy Bridge processors”, Third International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC13), Denver, CO, Nov. 17-22, 2013

Tan Nguyen, Daniel Hefenbrock, Jason Oberg, Ryan Kastner and Scott Baden, ”A software-based dynamic-warp scheduling approach for load-balancing the ViolaJones face detection algorithm on GPUs, Journal of Parallel and Distributed Computing”

Daniel Hefenbrock, Jason Oberg, Nhat Tan Nguyen Thanh, Ryan Kastner and Scott Baden, ”Accelerating Viola-Jones Face Detection to FPGA-Level using GPUs,” 18th Ann. Intl. IEEE Symp. on Field-Programmable Custom Comput. Machines, Charlotte, NC, USA, May 2-4, 2010, 8 pp

Tan Nguyen and Scott Baden, ”Automating the Communication-computation Overlap with Bamboo”, MiniSymposium at the SIAM conference on Computational Science and Engineering, Boston, MA, Feb 2013

ABSTRACT OF THE DISSERTATION

Bamboo: Automatic Translation of MPI Source into a Latency-Tolerant Form

by

Nhat Tan Nguyen Thanh

Doctor of Philosophy in Computer Science

University of California, San Diego, 2014

Professor Scott B. Baden, Chair

Communication remains a significant barrier to scalability on distributed-memory systems. At present, the trend in architectural system design, which focuses on enhancing node performance, exacerbates the communication problem, since the relative cost of communication grows as the computation rate increases. This problem will be more pronounced at the *exascale*, where computational rates will be orders of magnitude faster than that of the current technology. Communication overlap is an efficient method to hide communication by masking it behind computation. However, existing overlapping techniques not only require significant programming effort but also complicate the original program.

This dissertation presents a source-to-source translation framework that can realize communication overlap in applications written in MPI, a standard library for distributed-memory programming, without the need to intrusively modify the source code. We explore a strategy based on re-interpreting MPI, which executes the application under a data-driven model that can hide communication overheads automatically. We reformulate MPI source into a task dependency graph representation, in which vertices represent tasks containing computation code and edges represent data dependencies among tasks. The task dependency graph maintains a partial ordering over the execution of tasks, enabling the program to execute in a data-driven fashion under the guidance of an external runtime system. To automate the code translation process, we develop Bamboo, a source-to-source translator. Bamboo supports a rich set of MPI routines, including point-to-point, collective, and communicator splitting operations.

We show that, for a variety of applications, Bamboo is able to hide communication overheads on a wide range of platforms including traditional clusters of multicore processors, as well as platforms based on accelerators (NVIDIA GPUs) and coprocessors (Intel MIC). Specifically, we translate applications taken from three different application motifs: dense linear algebra, structured and unstructured grids. In all cases, Bamboo significantly reduces communication delays while requiring only modest amounts of programmer annotation. The performance of applications translated with Bamboo meets or exceeds that of labor-intensive hand coding. The translator is more than a means of hiding communication costs automatically; it also serves as an example of the utility of semantic level optimization against a well-known library.

Chapter 1

Introduction

Code optimization is always a costly phase in software development, and optimizing High Performance Computing (HPC) applications is no exception. At present, distributed-memory systems have evolved to a sophisticated level that requires applications to be heavily optimized to harness all resources provided by the hardware. We argue that the overheads of communication, especially off-node communication, have to be carefully addressed in tandem with improvements in computational rates. The reason is as follows. The programmer tends to uncover as much parallelism in the application as possible to leverage all computational capability of the system. In addition, the programmer generally can manage the memory hierarchy to take advantage of data locality, thereby mitigating on-node communication. However, the effect of these optimizations is to increase the relative overhead of off-node communication due to the following reasons. First, on-node optimizations normally do not reduce the amount of off-node communication. Second, the entire system is mostly idle when nodes communicate. Thus, the lower the cost of computation the higher the cost of communication, relative to the total execution time. Another motivation for tolerating communication delays is that computing performance has been growing faster than bandwidth and latency for several years. Indeed, Top500 lists [6] over the past few years indicate that the peak performance of the fastest system doubles every year, whereas bandwidth and latency can only double every 3 to 4 years [7]. With the current technological trend focusing on node architecture [8–10], i.e. enhancing computing nodes with accelerators such as Graphics Processing Units and Intel's Many-Integrated Core, this gap is expected to grow. Thus, tolerating communication delays is increasingly important, given that the ratio of communication (in bytes) over computation (in flops) is independent of hardware.

MPI, the de facto standard library for distributed-memory programming, has been used for more than 20 years [1] to develop applications running on high-end systems. However, MPI applications are often written under the Bulk Synchronous Parallel model [12], which does not offer support for writing latency-tolerant code. *Split-phase coding* [13–15] is a common technique for masking communication overheads under useful computation. This technique divides computation into smaller pieces, some of which are independent of current communication. Thus, the programmer can schedule independent computation and communication at the same time. Although overlapping communication with computation increases the performance by at most a factor of 2, it can provide other significant benefits when applied at scale (e.g. energy consumption). Implementing overlap via the *split-phase coding* technique, however, requires aggressive modifications to application data structures and control flow. This requirement poses a significant burden on the programmer, who may be a user focusing on domain science.

Source-to-source translation can be an efficient means of automating complicated optimizations on communication. For example, *Physics* [16] is a translation framework that allows the programmer to extend serial C applications to generate multiGPU code. This framework is designed to support only stencil computations [17–20], so the communication pattern is known beforehand. This knowledge enables the translator to generate *split-phase* code that overlaps communication with computation. In particular, the translator splits the computation into 2 parts: the first part performs stencil operations

[1] Message passing has been used for more than 30 years [11]

at the inter-process boundaries that depend on remote data, whereas the other works on an *inner region*, which is independent of communication. Therefore, the latter portion can be scheduled in parallel with the communication phase [21]. Similar to *Physics*, many other prior works that automate the communication overlap are often ad-hoc and thus does not support a wide range of applications [22–24].

1.1 Research contributions

The goal of this dissertation is to develop a source-to-source translation technique to address obstacles to realizing communication overlap on high-end systems. As opposed to previous approaches, the technique presented in this dissertation is a general-purpose translation framework that transforms MPI applications previously written under a synchronous programming model into a data-driven form that can overlap communication with computation automatically. In addition, unlike other direct approaches which offer an explicit data-driven execution model [25, 26], we exploit the information of communication operations embedded in an MPI program to reason about the data dependencies among processes. Incorporating such knowledge with a modest amount of user annotation, we are able to reformulate MPI source into a task dependency graph representation. Such a graph maintains a partial ordering over the execution of tasks of the graph, enabling the program to execute in a dataflow-like fashion under the guidance of an external runtime system.

This translation framework, which we call *Bamboo*, includes a programming model and a source-to-source translator. Bamboo employs a directive-based programming model, enabling the programmer to quickly annotate their code. The directive-based approach isolates software development from technological changes and allows the original code to be maintained and studied in a familiar form. Given an annotated MPI program, the Bamboo translator reformulates the source code to represent the application in a task dependency graph form. The Bamboo software stack comprises 2 layers: *core message passing* and *utility layers*. The *core message passing* layer transforms a minimal subset of MPI routines, whereas the *utility* layer implements MPI high-level routines by translating them into their point-to-point components, which will be then translated by the *core message passing* layer. Such a multi-layer design allows one to customize the implementation of MPI high-level routines.

We demonstrated that Bamboo is able to improve performance on a wide range of applications taken from three important application motifs [27]: dense linear algebra (matrix multiplication and LU factorization), structured grid (a Jacobi solver for Poisson's equation and a geometric multigrid solver for Helmholz's equation) and unstructured grid (a hydro solver to the Sedov blast wave problem [28]). Moreover, we validated Bamboo on a variety of computational platforms. In particular, we ran experiments at scale on CrayXC30 and CrayXE6 systems at NERSC (National Energy Research Scientific Computing Center). We also tested Bamboo on advanced node architectures, which accelerate node performance by offloading compute-intensive kernels to devices such as NVIDIA's GPU and Intel's MIC. We compared Bamboo against the basic MPI and hand optimized code variants. In all cases Bamboo realized a significant reduction in communication delays of the basic MPI variant with only modest amounts of user annotation. The performance of Bamboo meets or exceeds that of the hand optimized code variant.

We also noticed that task scheduling plays an important role in balancing irregular workloads arising in some applications. Bamboo supports non-preemptive task prioritization, enabling tasks of the graph to work more cooperatively, thereby improving the performance. In particular, when task prioritization is enabled, tasks can volunteer to yield the processor at the time of their choosing. Thus, hardware resources can be efficiently shared among tasks at possibly lower scheduling costs (e.g. task switching cost) than using a preemptive scheduling system. We validated this support using two applications taken from the dense linear algebra motif: SUMMA matrix multiply [2] and matrix factorization [56]. Experimental results demonstrated that we gained significant performance benefits by employing simple prioritizing algorithms.

1.2 Dissertation outline

The remainder of the dissertation is organized as follows. Chapter 2 reviews background on parallel programming models. Chapter 3 presents the Bamboo source-to-source translation framework. Chapter 4 presents the design and implementation of Bamboo's source-to-source translator. Next, chapters 5, 6, and 7 show experimental validation using applications taken from 3 HPC motifs: dense linear algebra, structured, and unstructured grids. Chapter 8 presents the support of Bamboo on advanced node technologies. Finally, chapter 9 concludes the dissertation and presents future work.

Chapter 2

Background

2.1 Overview

A programming model abstracts the underlying hardware and software architecture of a computing system and provides the programmer with high-level functionality, which is likely to be difficult to implement directly with low-level primitives. For example, a multiprocess-based programming model may provide a high-level data motion routine to transfer data from one process to another, hiding the low-level protocol controlling the data transfer that is transparent to the programmer. This chapter reviews parallel programming models, which are useful to understand discussions on source-to-source translation techniques presented in the remainder of the dissertation.

2.2 Synchronous model

2.2.1 Bulk Synchronous Parallel

BSP (Bulk Synchronous Parallel) [12] is an abstract model for parallel systems, including both shared-memory and distributed-memory platforms. A program written under the BSP model executes as a set of processes, which proceed through a sequence of *supersteps*, each consisting of computation, communication and a barrier synchronization as shown in Fig. 2.1. Processes communicate with each other explicitly. At any given time, all processes are either computing or communicating, but not both. Two consecutive *supersteps* are separated by a barrier synchronization point. Data communicated during one *superstep* becomes visible during the following step.

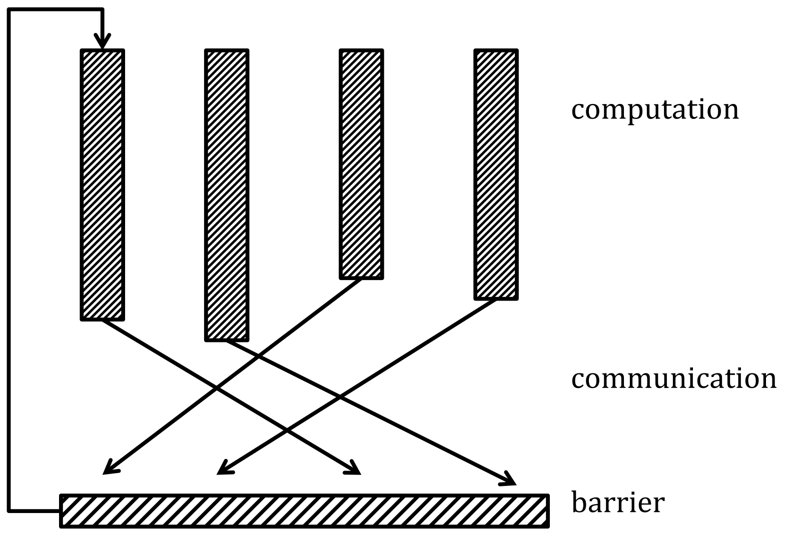


Figure 2.1. A BSP program consists of multiple processes, executing *supersteps*. Within each *superstep*, processes compute, communicate, and synchronize. The communication among processes is explicit.

Performance modeling under the BSP model is simple due to the barrier synchronization [12, 29]. In particular, the cost of a BSP program can be calculated by accumulating the cost of each *superstep*. The running time of a *superstep* is expressed in terms of a few parameters as follows:

$$T_{superStep} = w + gh + l \tag{2.1}$$

where

- w: the maximum time to perform local operations by any processor
- h: maximum number of messages sent or received by any processor
- g: 1/g is the available bandwidth per processor

- l: overhead of the global synchronization

BSPlib [30, 31] is a communication library for BSP programming. The library contains a routine to separate *supersteps*. In addition, it provides communication routines, including both one-sided communication using remote direct memory access (RDMA) and two-sided message passing communication.

PVM (Parallel Virtual Machine) [32] and MPI [33] (Message Passing Interface) are two common libraries that can be used to develop BSP programs. PVM abstracts all heterogeneous hosts into a single virtual machine. Based on this virtual machine, PVM provides a programming interface and runtime support to develop and execute applications.

MPI aims to provide a standard interface for inter-process communication. The primary goals of MPI are portability and performance. Beside point-to-point primitives, MPI also provides communicator for process grouping and collective operations. We will be presenting more details about MPI in the next chapter.

2.2.2 LogP

LogP [1] is a parallel model that describes a parallel architecture using four parameters: latency (L), overhead (o), gap (g), and number of processors (P). Under the LogP model, a processor can be either *stalling* or *operational*. A processor is *stalling* when it is submitting a message to the communication medium. o is the number of clock cycles required by the processor to prepare for a message. This processor becomes *operational* when the submitted message is accepted by the medium. When *operational*, a processor can perform computation or submit/acquire a message. The behavior of the communication medium is modeled by two parameters g and L. In particular, g clock cycles must elapse between consecutive submissions or consecutive acquisitions by the same processor. In addition, the communication medium can deliver a message to its destination at most L clock cycles after its acceptance. The LogP model assumes that the network has a finite capacity. Thus, it requires that at any time there are no more than L/g messages in transit for the same destination. Fig. 2.2 shows a tree-based implementation of the broadcast operation. The performance is predicted using the LogP model.

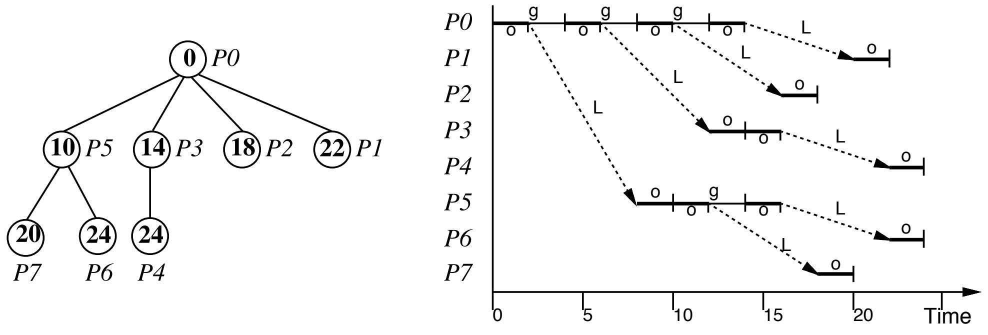


Figure 2.2. Modeling the performance of a tree-based broadcast operation using the LogP model [1]

BSP and LogP are substantially equivalent since they can efficiently simulate the other. In particular, BSP can simulate LogP with constant slowdown and LogP can simulate BSP with at most logarithmic slowdown [29]. Though LogP is more descriptive, BSP is preferable due to its simplicity and portability.

2.3 Partitioned Global Address Space

Due to the global address space, some programmers prefer to retain the shared-memory interface even when the underlying architecture is distributed-memory. Doing so, however, compromises performance due to the lack of locality and poor performance of short messages. PGAS languages (Partitioned Global Address Space (PGAS)) offer a shared-memory like programming interface but add control over data layout to maintain performance.

Unified Parallel C (UPC) [34] is a parallel extension to C, and is perhaps one of the most popular PGAS languages. Under UPC, each program spawns a fixed number of

UPC threads, and every thread runs the same code in a SPMD manner (Single Program Multiple Data). Each thread has private space for local memory and shared space for a partition of the global memory. The task of a UPC programmer is to explicitly control this data layout, though the mapping of threads and their data onto physical processors is handled by the UPC runtime system. In addition, a pointer or reference to shared space must be statically distinguishable from those to private space. UPC employs a *1-sided* communication model, and there is no explicit send-receive matching (e.g. as in MPI). Thus, data communication can be processed without any matching and synchronizing overhead. This feature is important for fine-grained communication; however, it does not benefit long message transfers. Beside UPC, there are also many other PGAS languages. Co-array Fortran [35] and Titanium [36] are extensions to Fortran and Java, respectively. Although each of these languages is influenced by their base serial language, they share many similar features. Particularly, these languages are SPMD and support static parallelism, meaning that the number of threads is fixed during the course of computation. Again, references to private and shared spaces must be statically distinct. This feature is to avoid expensive memory checks when a local pointer is dereferenced.

HPCS (High Productivity Computing System) languages such as Fortress [37], Chapel [38], and X10 [39] are PGAS languages that aim to exploit emerging high performance computing systems. Whereas X10 is a parallel extension to Java, Fortress and Chapel are not extensions of any existing serial language. Rather, these two languages provide abstractions to describe high-level parallelism. Fortress also employs a hierarchical notion of partitioning to better support hierarchical parallelism and hardware. Compared to the traditional PGAS languages, HPCS languages provide dynamic parallelism and more sophisticated synchronization mechanisms. For example, a Fortress program consists of a set of threads (*activities* in X10) and *memory locations* (*places* in X10 and *locales* in Chapel). Threads are spawned on the fly, hence the term *dynamic par-*

allelism. In multithreading models and traditional PGAS languages, threads coordinate via a global barrier. To better support synchronization under *dynamic parallelism*, X10 provides a clock. A clock is basically a barrier, but is deadlock-free and may consist of a subset of threads.

2.4 Dataflow

Dataflow has been used in compilers to achieve implicit instruction-level parallelism [40–42]. For example, in 1967 Tomasulo proposed a dynamic scheduling algorithm (AKA Tomasulos algorithm) to enable out-of-order instruction execution. This algorithm first eliminates anti-dependences between instructions through renaming. It then maintains and schedules a dependency graph of instructions based on the availability of operands and functional units.

Dataflow was then found to be applicable to user-level parallelism. Blumofe et al. presented Cilk [43, 44], an extension to C that employs user-level dataflow. Under Cilk the programmer specifies application parallelism using keywords such as *spawn* and *sync*. It is the responsibility of Cilk to dynamically assign computational tasks to processors. Internally each Cilk program is represented as a DAG (directed acyclic graph) that can unfold as the program executes. This DAG is a control-flow and dataflow graph where each vertex is a sequence of instructions that do not contain Cilk keywords and edges represent ordering constraints between vertices. One of the most interesting ideas in Cilk is the *work-stealing* concept. A processor that runs out of work can steal computational tasks from the task queue of another *victim* processor. This mechanism balances the workload among processors. Internally each processor maintains a double-ended queue of suspended tasks such that the dequeue operation can work in both ends. The processor that owns the queue can insert and remove tasks from one end while other processors can steal tasks from the opposite end. Blumofe et al. also presented a performance model to predict the execution time of a Cilk program. This model is based on the concepts of T_1 and T_∞ , where T_1 is the total work and T_∞ is the critical path. Given the maximum performance on p processors provided by a greedy scheduler $T_p = T_1/p + T_\infty$ [45], the authors introduced an empirical performance model as the following: $T_p = T_1/p + c_\infty T_\infty$ where c_∞ is a small constant. Cilk performs well on shared-memory architectures and has been commercialized to Intel Cilk Plus. However, it is costly to support Cilk on distributed-memory architectures due to high overheads of dynamic scheduling and task migration on multiple nodes.

PLASMA [46] (Parallel Linear Algebra for Scalable Multi-core Architectures) is a library for solving dense linear algebra problems on systems of multicore processors. Unlike synchronous implementations of dense linear algebra routines such as LAPACK, PLASMA hides communication overheads by dynamically scheduling a task graph. To explore parallelism, the graph structure unfolds dynamically during execution. However, due to resource constrain, PLASMA handles only a small portion of the graph at a time using *sliding window*. PLASMA employs QUARK [47], a runtime system built on the POSIX thread library, to schedule the task graph.

2.5 Actors

Instead of specifying a list of actions that each task has to do, *actors* models consider a task as an object called *actor* [48, 49]. Each *actor* has its own state, and it can send messages to other *actors*. In response to a message, an *actor* can change its state, send messages, terminate, and even create other *actors*. A useful observation is that *actors* models can support communication-computation overlap with the help of virtualization. In particular, when there are more *actors* than physical processors, the dependency between *actors* can be represented by messages. A scheduler will select an *actor* that satisfies all dependencies to execute when other *actors* are communicating.

Charm++ is a distributed-memory programming model based on *actors* [50]. Under Charm++, *actors* communicate with each other by calling methods of their partners. These methods are asynchronous and can be invoked from *actors* on remote processors. Charm++ offers communication and computation overlap by virtualization. Specifically, the number of *actors* will be larger than the number of processors. A runtime system will schedule *actors* based on the availability of processors and necessary data. Further, Charm++ supports load-balancing by allowing task migration.

Tarragon [52–54] is an *actor-like* programming model, therefore it can support communication-computation overlap. Each object in Tarragon is called a task, which can exchange messages with other tasks. Tasks are connected through edges to create a task-dependency graph. Each task has a local state which is recognized by the runtime system. Once all messages are received, the task can be scheduled to run. An interesting feature of Tarragon is that it supports iterative programming decomposition. This technique can help Tarragon reuse space, which increases the scalability of the model. Note that large projects can have up to millions of lines of code, which is not practical to build a full dataflow graph for these programs.

2.6 Summary

Table 2.1 summarizes the programming models presented in this chapter. We classify these models based on a few criteria: memory address space, communication model, and latency hiding support.

BSP based models such as BSPlib, PVM, and MPI require the programmer to explicitly program the movement of data among processes. Such requirement is to leverage locality, enabling the high efficiency of local computation. In addition, these models allow the programmer to easily analyze the performance due to the synchronous execution model. However, the drawback of the synchronous execution model is that it does not support automatic latency hiding.

PGAS based models provide a global address space that allows the user to program with a shared memory interface regardless the underlying platform. However, these models retain the explicit communication model to allow the programmer to take advantage of locality (e.g. get() and put() in UPC). Thus, an efficient program written in a PGAS based model resembles a program written under the BSP model. A notable feature of PGAS based models is that they support the one-sided communication model. Compared to the two-sided communication model, this model reduces the communication startup overhead. However, this advantage is at the cost of more complicated synchronization (e.g. lock, semaphore) which must be written manually by the programmer. In addition, PGAS based models do not support automatic latency hiding.

Under dataflow and *actors* models, the programmer does not need to explicitly handle the data communication. Instead, the programmer provides information about the data dependency among tasks/*actors*. A runtime system is responsible for handling communication and scheduling the execution of tasks/*actors* depending on the availability of data. Thus, these models automatically support latency hiding by scheduling computation and independent communication at the same time.

Table 2.1. Characteristics of different programming models (SM: shared memory, DM: distributed memory)

Name	Model	Address Space	Data Communication			Latency
			2-sided	1-sided	implicit	Hiding
BSPlib	BSP	DM	✓	✓		
PVM	BSP	DM	✓			
MPI	BSP	DM	✓	✓		
UPC	PGAS	DM		✓		
Fortress	PGAS	DM		✓		
Chapel	PGAS	DM		✓		
X10	PGAS	DM		✓		
Cilk	Dataflow	SM			✓	✓
PLASMA	Dataflow	SM			✓	✓
Charm++	Actors	DM			✓	✓
Tarragon	Actors	DM			✓	✓

Chapter 3

Bamboo

3.1 Overview

This chapter presents the code translation framework provided by Bamboo. Similar to other automation initiatives, Bamboo aims to produce delay-tolerant code that automatically hides communication overheads by overlapping communication with computation. However, unlike other approaches which directly offer a data-driven programming model, Bamboo exploits the information of communication operations embedded in an MPI program to reason about the data dependencies among processes. By incorporating such knowledge with a simple, directive-based programming model, Bamboo is able to reformulate MPI source into a task dependency graph representation. The task dependency graph maintains a partial ordering over the execution of tasks of the graph, enabling the program to execute in a data-driven fashion under the guidance of an external runtime system. This approach alleviates the effort that would otherwise be needed to migrate extremely large legacy code bases to a communication-tolerant form. In addition to the programming model, the Bamboo translation framework also includes a communication model that specifies how data is exchanged among tasks and an execution model that describes task execution behavior.

3.2 MPI: challenges in hiding latency

3.2.1 Bulk synchronous MPI

An MPI program executes as a group of processes, each assigned a unique rank. MPI processes work on disjoint partitions of the application's data. In order to easily manage the set of processes, MPI introduces the concept of a *communicator*. Upon the initialization of the MPI environment (i.e. via MPI_Init), a global, pre-defined communicator called *MPI_COMM_WORLD* is constructed. This communicator contains all processes of the program, which can be then optionally split into smaller communicators. During the course of execution, MPI processes explicitly communicate with each other via messages. MPI programmers rely on *send* and *receive* primitives to express the inter-process communication. Message passing algorithms must obey a matching rule as follows. Whenever a process posts a send request, exactly one process must post a receive request matching that send. In addition, messages between a given sender and a receiver will not overtake, meaning that they will arrive in the order that they are sent. This model is often referred to as a *two-sided* communication model as it requires both sender and receiver to be involved in the communication process. MPI also offers a *one-sided* communication model, which excludes the receiver from the communication. However, this dissertation considers only the *two-sided* communication model as it accounts for the vast majority of MPI applications.

To support communication operations involving multiple processes of a communicator, MPI introduces the concept of a collective. A collective operation requires all processes within the communicator to participate. A collective routine also specifies the input and output of each individual process participating in the operation. MPI includes a plentiful set of collective routines in order to capture a variety of collective communication patterns arising in practice. The algorithm for each collective routine, however, varies among MPI implementations. Thus, MPI programmers may also use send and receive primitives to implement their own collective routines with algorithms customized for their specific applications.

Recall from Section 2.2 that Bulk Synchronous Parallel (BSP) [12] is a programming model commonly used for developing MPI applications. We call MPI programs written under the BSP model *bulk synchronous MPI*. Such MPI programs execute a sequence of computation and communication phases. The effect of the bulk synchronous execution model is that, at a particular time, each MPI process performs either communication or computation, but not both simultaneously. Specifically, in the communication phase the program execution at the receiver site blocks until the expected data has arrived. Thus, BSP can incur significant performance penalties, as processors sit idle during the communication phase. Nevertheless, this model has been widely used for decades due to its simplicity in programming, and in analyzing and predicting the performance.

We anticipate that MPI will continue to play an important role in the future of high performance computing due to the following reasons. First, MPI enables the programmer to express data locality explicitly, which has been proven to be extremely crucial to realizing high performance at scale. Second, the well-defined set of communication routines included in MPI is always attractive to the programmer due to its completeness and usability. However, the future of high performance computing, e.g. exascale computing, exposes many challenges for which MPI applications written under the BSP model have to be highly optimized or rewritten in a latency-tolerant form. One of the challenges is that the performance gap between compute nodes and the interconnection network is increasing. As a consequence, communication can become a bottleneck when computation no longer dominates. Another challenge is that strong scaling will be more commonplace at exascale since memory capacity is not expected to continue to grow as it has been in the past, due to a power consumption constraint. Under strong scaling, a fixed problem size is maintained as the number of processes increases. Thus, the relative overhead of communication can quickly grow as the computing time decreases.

3.2.2 Split-phase coding

Hiding communication overheads by overlapping with computation is an effective means of tolerating communication delays lying on the critical path of a *bulk synchronous MPI* program. *Split-phase* coding is a well-known optimization technique for overlapping communication with computation [13–15]. To enable this optimization, the programmer needs to identify computations that can be split into smaller parts, some of which are not dependent on communication. This technique exposes more parallelism so that the communication among MPI processes can be hidden. Once the computation has been split, the programmer must also statically co-schedule communication with portions of the computation that are independent of communication. Writing *split-phase* code is challenging for the following reasons.

1. In most applications, it is nontrivial to identify finer-grained computations that do not depend on communication. In addition, exploiting fine-grained parallelism improperly may hurt locality, which in turn reduces computing rates.

2. The programmer needs to restructure the code to enable communication to be pipelined with computation. Restructured code is both complicated and error-prone. As a result, *split-phase* coding complicates any efforts to develop and optimize the application further.

High Performance Linpack (HPL) benchmark [55–57] is a typical example that shows the challenge of applying the *split-phase* coding technique. HPL is a direct solver for systems of linear equations using the Gaussian elimination method. The most important kernel of HPL is LU factorization, which decomposes an input matrix into lower- and upper-triangular matrices. The input matrix is first partitioned into small submatrices called panels, which are then distributed to a two-dimensional grid of MPI processes in a cyclic fashion.

To overlap computation with communication, HPL splits the computation into 2 parts: the first part depends on the arrival of a remote panel while the other does not. HPL then starts the communication of the following panels while at the same time performs the second part, which is indepedent of the communication. Embedding such a scheduling policy into an MPI program complicates the code structure, slowing down the application development and maintenance. In order to enhance the chance to overlap, MPI processes also have to frequently poll for the arriving of messages at many places in the program in order to forward arrived messages to neighbors in a timely manner. In addition, since each process is responsible for both performing computation and handling communication, HPL has to further splits the computation into smaller chunks so that the MPI process can quickly response to the arrival of messages. We deem that such a polling algorithm makes the code much more difficult to read. It is a strong evidence to show that communication policies shouldn't be embedded in an MPI program. Instead, they should be handled by an independent communication handler.

We next present a novel approach taken by Bamboo, which aims to produce latency-tolerant code without complicating the application software.

3.3 Bamboo programming model

3.3.1 Bamboo's approach

Bamboo does not use *split-phase* coding. It employs a different approach to tolerate communication overheads. First, Bamboo factors work scheduling out of the program execution. It reformulates MPI source into a form so computation can be efficiently scheduled by an external runtime scheduler. In fact, Bamboo reformulates MPI source into a new representation called task dependency graph, which will be presented in detail in the next part of this section. At the high level, such a graph maintains a partial ordering over the execution of tasks of the graph, enabling a runtime scheduler to schedule tasks based on the availability of data. Second, Bamboo factors out communication decisions as follows. The MPI application contains only fundamental knowledge about the flow of data among processes. Once this basic MPI application is rewritten under the form of a task dependency graph, it is the responsibility of runtime communication handlers to service the communication among tasks. In particular, a communication handler at the sending site may buffer messages before sending them into the network. At the receiving site, a communication handler is listening for incoming messages, and this handler may buffer arrived messages before injecting them into tasks. In addition, communication handlers may need to make routing decisions if each compute node is connected to the interconnection network via multiple end-points. Details of the services that a task dependency graph expects from a communication handler will be presented at the end of this chapter.

Decoupling scheduling and communinication decisions from software development is an appealing approach since it helps isolate the application from future technological changes in node architectures and interconnection networks. However, rewriting MPI application from scratch, in the form of a task dependency graph requires significant effort from the programmer. To deal with this drawback, Bamboo employs a source-to-source translator that automates the code transformation process. The translator treats MPI as an embedded domain specific language and MPI data types and methods as language primitives. Although most of the transformation phases are fully automatic, the translator requires a modest amount of user annotations in order to generate high quality code. Bamboo introduces a programming model, including the definition of a task dependency graph and the description of a programming interface upon which the programmer relies to annotate MPI code. In the remainder of this chapter, we present the programming model provided by Bamboo. Implementation details of the source-to-source translator will be presented in the next chapter.

3.3.2 Task dependency graph

Task and dependency

The code produced by Bamboo is a Task Dependency Graph (Task Graph for short) representation in which vertices are tasks containing computation code and edges represent data dependencies among tasks. Incoming edges of a task represent input data needed for task execution, whereas outgoing edges correspond to output data produced by task computation. Tasks do not share any data. Rather, they work on disjoint data, and the edge is the only means for communicating data among tasks. There is no restriction on the number of tasks and edges in a graph, though a graph instance maintains a fixed number of tasks throughout its lifetime. Edges, however, may change during the execution of a graph. At a particular time, edges represent the data requirement of a sequence of code that the task is about to execute.

Figure 3.1 shows a snapshot of a task graph instance consisting of 16 tasks. Edges represent the data dependencies among tasks at the moment, which may be changed later on. In this snapshot, we note that there is a task containing 2 incoming edges from another task. It is totally legal to have, at a time, two or more edges connecting two tasks, as edges can be labeled as shown in Figure 3.1. This notation is useful when a task simultaneously requires multiple inputs from another task. Of course, such edges may be safely fused into a single edge representing for the aggregation of individual inputs from the same task. Likewise, a single edge may be dynamically split into smaller ones, each assigned a different label.

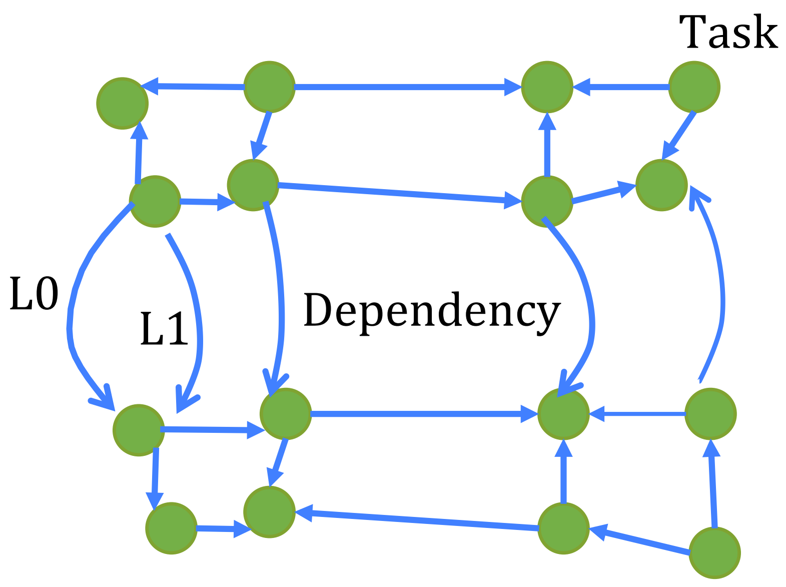


Figure 3.1. A task dependency graph snapshot. Vertices are tasks and edges are data dependencies among tasks. If there exist two or more edges between two tasks, edges will be labled.

Task state

Each task is augmented with a *state*, which is visible to the scheduler. The scheduler can use state to make scheduling decisions. Only a task can modify its state, and a task is not able to directly modify the state of another task. In particular, the state alternates between *WAIT* and *EXEC* during the lifetime of a task. When a task needs to wait on data from others, it changes the state into *WAIT*. When an input is available, task may change its state to *EXEC*. A task is called *runnable* if its state is *EXEC*, as it is an indication that the scheduler can pick the task for executing.

Graph-based, data-driven execution

The execution of a task graph is driven by the availability of data as follows. Tasks are *runnable* (i.e. *state* = *EXEC*) if they have received all input data and are inactive (i.e. *state* = *WAIT*) otherwise. *Runnable* tasks can line up to be executed. During the execution of *runnable* tasks, communication can be handled by independent communication handlers. Thus, automatic latency hiding can be realized by scheduling a task dependency graph. Neither task scheduling nor communication handling is a part of the graph. Rather, they can be treated independently of the task definition. This is true even in the case of irregular applications, where computation and communication vary among tasks. Indeed, irregular tasks can simply be augmented with meta-data, high-level discriptions that help guide an external scheduler towards intelligent scheduling decisions. Details of how a task graph executes will be presented in the Bamboo execution model at the end of the chapter. We next present how a task graph can be constructed from an MPI program.

3.3.3 Reformulating MPI source into a task graph form

Given an MPI application programmed under the BSP model, one may wonder if there exists a formula to construct the task graph described previously. A task graph consists of tasks and edges. Thus, constructing a task graph entails two phases: i) defining the task and ii) defining the data dependency.

Task

MPI applications employ processes to partition a global problem across processing domains. Under the BSP model, processes hold processing resources during the whole program lifetime, even while waiting for data. Such an execution model prevents communication from being overlapped with computation. In order to hide latency, Bamboo employs a task-based, time-sharing execution model such that some tasks can execute on the processors while communication is executing simultaneously.

Process virtualization (AKA *oversubscription*) is a common technique to realize task parallelism from a process-based parallel program [58, 59]. This technique applies the same work-partitioning pattern employed in the original parallel program, but at a finer granularity. Figure 3.2 shows how MPI processes can be virtualized to run with multiple tasks per MPI process. As in the original MPI program, a graph consists of a fixed number of tasks upon construction, each assigned a task ID. The amount of work assigned to a task is then based on its unique ID and the work-partitioning algorithm used in the original MPI program. Communication patterns among MPI processes remain unchanged under process virtualization.

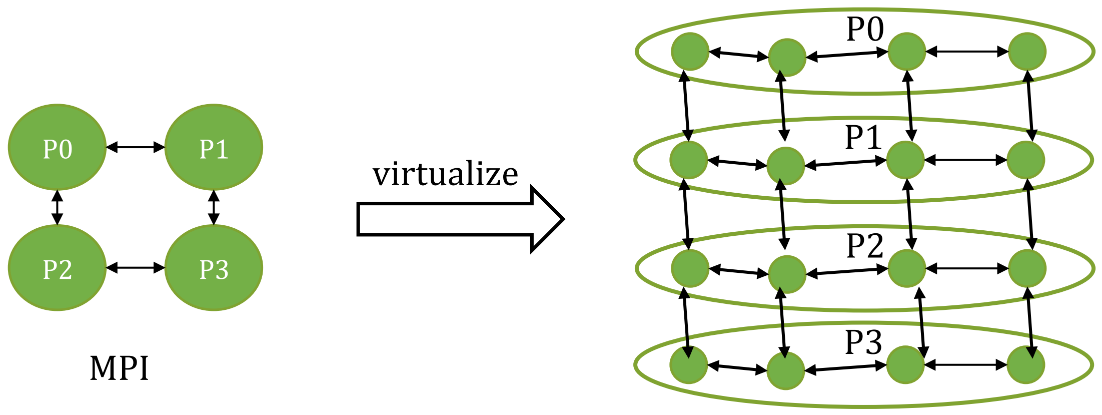


Figure 3.2. Virtualizing MPI processes to finer-grained tasks. Communication patterns remain unchanged under the virtualization technique.

In virtualizing MPI processes, Bamboo may also realize an additional benefit in improving cache locality. This additional benefit makes it difficult to evaluate the individual benefit of hiding latency. Cache blocking is a well-known optimization technique to improve locality. Instead of fetching the entire contiguous row (row major) or column (column major) of an array, this technique loads a small 2D or 3D block at a time so that loaded data can be heavily reused. The parameters of a blocking algorithm include the cache size, which varies with memory subsystems, and the stride access which depends on both application and problem size. Since the cache size and stride access often vary, implementing the blocking optimization is challenging. Tasks of a graph, on the contrary, own smaller portions of the global data. The higher number of tasks the finer data partitions owned by tasks. Thus, if the cache blocking optimization is not present in the original MPI program, cache locality is automatically obtained when we use a substantial number of tasks.

An important question then arises is that whether one could estimate the individual benefit of hiding latency based on profiling measurements at the task level only. In addition, such an estimation should be independent of the effect on cache locality. If this is possible, which metric should be used to evaluate the overlapping technique? To answer these questions, consider a queuing system as follows. For each process, one can configure a queue of *runnable* tasks. The size of the *runnable* task queue is bounded by the number of tasks per process. Tasks in the queue will be eventually served by a *worker thread*. Tasks arrive in the queuing system once they have received all data needed to become *runnable*. Tasks leave the system when they need more data to continue the execution. Equation 3.1 presents Little's law, which establishes a relation between the system occupancy and the averaged number of *runnable* tasks per second, given the response time.

$$Occupancy = RequestRate * RepsonseTime \tag{3.1}$$

where *occupancy* represents the averaged workload assigned to the *worker thread* relative to the peak amount of workload that it can handle, *RequestRate* is the rate that tasks become runnable, and *ResponseTime* is the time needed to execute a task.

It can be seen that one can measure *RequestRate* and *RepsonseTime* experimentally by integrating a *performance profiler* into the task scheduler to collect the task arrival rate and the response time of the *worker thread*. To illustrate the priciple, we consider a queuing system with 10 tasks/process, where each task requires 0.001 seconds to complete one iteration. These 10 tasks enter, leave, re-enter the system and so on. If the *performance profiler* observes only 900 *runnable* tasks/second, the occupancy is 900 x 0.001 = 90%. Thus, each task needs to become *runnable* at least 1000/10=100 times per second to yield 100% occupancy. An intuitive explanation for this number is that the queue must receive at least one *runnable* task every 0.001 second so that it can keep the *worker thread* busy. We can see that, without communication, each task can be served by the *worker thread* at the rate 1/0.001 = 1000 times per second. Thus, the remaining 900 times can be used for tolerating the communication overheads without any loss in performance.

Little's law is also useful in performance modeling, where we are also interested in *IdleTime*, which is the interval that starts when a task leaves and ends when it re-enters the system. RequestRate may depend on the *IdleTime* if the time for executing other *runnable* tasks is smaller than *IdleTime*. Consider, for example, a matrix multiplication operation, where each task owns NxN submatrices, performs $2*N^3$ floating point operations and moves $2*8*N^2$ bytes of double-precison data. Assume that the computing rate of a *worker thread* is 16 GFLOP/s and the bandwidth of remote memory access is 1GB/s. If the queuing system is configured with only one task per process, there is no other task to tolerate the *IdleTime*. Thus, RequestRate = 1/(ResponseTime + IdleTime). As a result, *Occupancy* is always less than 100% in this case. In addition, the higher the *IdleTime*, the lower Occupancy. Indeed, assume that $N = 2^8$, Response time in this case should be $2*N^3$/(16 GFLOP/s)= 2^{-9} seconds and *IdleTime* is roughly $2*8*N^2$/ (1GB/s) = 2^{-10}. Thus, the occupancy is $(2^{-9} + 2^{-10})^{-1} * 2^{-9}$ = 67%. When there are 4 tasks per process, the problem size of each task is smaller (2^7). The new Response time is $2*N^3$/(16 GFLOP/s)= 2^{-12} and the new *IdleTime* is $2*8*N^2$/ (1GB/s) = 2^{-12}. Thus, if one can sketch a scheduling plan (either algorithmicaly or by simulation) such that the *IdleTime* can be hidden by the computation of at least 1 task, RequestRate will be 1/ResponseTime, resulting in an 100% *Occupancy*.

While reformulating MPI processes into tasks is straightforward, reinterpreting

MPI communication calls as input and output is more involved. We next discuss how Bamboo defines the input and output of a task.

Input and output

The execution of a task comprises a sequence of phases, each consisting of 3 modules: *input*, *compute*, and *output*. Unlike MPI processes, which explicitly pass messages, tasks do not communicate directly. Instead, *input* and *output* specify a flow of data from one task to another as follows. *Input* defines the data that a task needs from others, as well as how and when such data can be safely injected into the internal data structures of the task so that it can proceed to the execution. *Output* defines the data produced by task's computations as well as when the data is ready to be sent to another task.

A question then arises is that whether or not the communication information embedded in an MPI program is sufficient to express *input* and *output*. Information that may be useful includes arguments for send and receive primitives such as message source and destination, message size, tag, etc. In addition, the circumstances under which the send and receive occur are also important. To answer this question, consider a simple MPI example shown in Fig. 3.3, a common implementation of MPI_Reduce based on a Binomial tree structure. Each of the P processes has a parent and log(P) children. Except for leaf nodes, processes have to wait for data from each child, compute a new localValue, and at the end send the final localValue to the parent.

It is likely that one will express each individual Recv as input and the following reduction operation as computation. This scheme, however, may not work efficiently since a task comprises just a few instructions to perform the reduction operation. After that, this task needs to yield the processor if the next input is not yet available (i.e. data from the next child). Task switches are much more costly than the reduction operation.

```
if(leafNodes)MPI_Send(localValue, parent)
else
{
  for (smallest child to largest child)
  {
    MPI_Recv (remoteValue, from a child)
    localValue = ReduceOp(localValue, remoteValue)
  }
  if(hasParent) MPI_Send(localValue, parent)
}
Computations...
```

Figure 3.3. A tree-based reduction, where each of the P processes has a parent and log(P) children. Leaf nodes simply send data to their parent. Other nodes receive data from each child, update local value, and send the final value to their parent.

Thus, the performance could be affected by the overhead of many task switches.

```
if(leafNodes)MPI_Send(localValue, parent)
else
{
  for (smallest child to largest child)
  {
    MPI_Irecv (&remoteValue[child], from a child)
  }
  MPI_Waitall(all children)
  for (smallest child to largest child)
    localValue = ReduceOp(localValue, remoteValue[child])
  }
  if(hasParent) MPI_Send(localValue, parent)
}
Computations...
```

Figure 3.4. Buffering messages to eliminate anti-dependencies between Recvs

Consider the improved implementation of reduction shown in Figure 3.4. We use a buffering technique that can eliminate the dependencies among MPI_Recv calls, thereby enabling multiple receive requests to happen at any order. If one express this program into a task graph, each task defines multiple inputs, each corresponding to Irecv a call. Once a task receives all inputs, it performs the reduction operations, makes output available, and continues the computations after the reduction without yielding the processor.

From this example, we can see that the programmer can express input in different ways, which result in various execution behaviors. Unfortunately, static analysis is unable to reason about the pros and cons of each option. In addition, modifying the MPI source code as shown in Fig. 3.4 should be avoided as it complicates the source code and may change the performance of the original program. As a result, Bamboo provides a programming interface based on user directives so that the programmer can design the input in their own way. With this programming interface, the programmer does not require to intrusively modify the MPI source code. Such programming model must also be simple enough so that the programmer does not feel like working with a direct data-driven programming model.

3.3.4 Programming interface

Bamboo presents a directive-based interface that allows the programmer to quickly annotate an MPI application without intrusively modifying the original program. To construct a task graph we first locate *evaluation points*, where tasks determine when they have received all inputs so they can execute. In addition, Bamboo also determines *input windows*, each covering a set of *inputs* that a task needs to check at an *evaluation point*. *Inputs* within an *input window* enable not only the computations within the window but all computations residing between the current and the next *evaluation points*. To support the extraction of *evaluation points* and *input windows*, Bamboo provides *code region*. Each *code region* defines an *evaluation point* and an *input window*, which we next present.

Olap-region

A Bamboo program is a legal MPI program, augmented with one or more code regions called *olap-regions* as shown in Fig. 3.5. An *olap-region* is a section of code containing communication to be overlapped with computation. *Olap-regions* can be nested. The entry into an *olap-region* is the *evaluation point* where a task either continues or it returns control to the scheduler because the required input data is not yet available. Receive operations residing within an *olap-region* will be included in the *input window* corresponding to the *evaluation point* of the olap-region. Bamboo preserves the execution order of *olap-regions*, which a task runs sequentially, one after the other. However, there is no implicit barrier at the exit of an *olap-region*. This allows a task to exit an *olap-region* and continue the execution until it meets the following *olap-region*, where it can enter as long as all inputs defined by the corresponding *evaluation point* and *input window* are ready.

```
#pragma bamboo olap
{
    ...
    #pragma bamboo olap
    {
    }
    ...
}
...Computations...
#pragma bamboo olap
{ ... }
```

Figure 3.5. An MPI program annotated with *olap-regions*

Communication blocks

Within an *olap-region*, send and receive calls are grouped in communication blocks. There are two kinds of communication blocks: *send* and *receive*. A *send* block contains *Sends* only. In most cases, a *receive* block contains *Recvs* only, except for the following situation. If a *Send* consumes data obtained from a prior *Recv* (read after write dependence), then it has to reside within an appropriate *receive* block, either the same block as the Recv, or a later one. Communication blocks specify a partial ordering of communication operations at the granularity of a block, including associated statements that set up arguments for the communication routines, e.g. establish a destination process, pack and unpack message buffers. While the statements within each block are executed in order, the totality of the statements contained within all the send blocks are independent of the totality of statements contained within all the receive blocks. This partial ordering enables Bamboo to reorder send and receive blocks. For example, Bamboo can move all send blocks up front and outside of the *olap-region*, enabling all outputs to be sent out to fulfill inputs from the current olap-region onwards. Bamboo does not reorder blocks of the same type. However, it is worth noting that inputs can arrive in any order, as they can be buffered upon arrival and then injected into task in the order specified by the programmer.

```
#pragma bamboo olap
{
    #pragma bamboo send
    { ... }
    #pragma bamboo receive
    { ... }
    #pragma bamboo send
    { ... }
    #pragma bamboo receive
    { ... }
    ...
}
```

Figure 3.6. Annotation showing a single *olap-region* with enclosed *send* and *receive* blocks

Computation block

In addition to *communication blocks*, an *olap-region* may contain a single computational block (AKA *compute* block). The *compute* block is optional and may contain other *olap-regions*. The introduction of *compute* block does not provide any performance benefit. Rather, it provides a means of merging 2 *olap-regions* as shown in Fig.3.7. The semantics of the *compute* block is as follows. The compute block depends on all prior communication blocks, and it has to be executed before successive communication blocks. This implies that Bamboo cannot reorder communication blocks across the *compute* block.

```
#pragma bamboo olap
{
  #pragma bamboo send
  { ... }
  #pragma bamboo receive
  { ... }
  ...
  #pragma bamboo compute
  {
    computations
    //May hold olap-regions
  }
  ...
  #pragma bamboo send
  { ... }
  #pragma bamboo receive
  { ... }
}
```

Figure 3.7. Optional *computation* block within an *olap-region*

3.3.5 Examples

We now present intuitive examples containing widely used communication patterns arising in MPI applications. These examples demonstrate how existing MPI applications can be transformed into equivalent task graphs with no aggressive code modification but only a modest amount of user annotations.

The first example is a Jacobi iterative solver for 3-dimensional Poisson's equation. This code repeatedly sweeps a 7-point stencil operator over a 3-dimensional mesh, each updating a data element in the mesh using 6 neighboring values. This code also frequently computes the residual error of the solution. The mesh update goes on until the residual error is lower than a small threshold or a pre-defined number of iterations have elapsed, whichever comes first. Owing to the dependency on nearest neighbor data, processes communicate with others at the interface of their respective subdomains. We annotate the MPI source code with an *olap-region* as shown in Fig. 3.8 (lines #4 to #24). The translation of *MPI_Allreduce* (line #27) will be handled automatically by Bamboo.

The *olap-region* annotated in the 3D Jacobi solver is for the 7-point stencil update, where processes communicate with nearest neighbors at every mesh sweep. Since each process sends and receives different data, these 2 activities are independent of each other. Thus, all MPI_Isend calls are grouped in a *send* block while all MPI_Irecv calls reside in a *receive block*. Each process can start the local grid update and grid swapping once it has received all data from its 6 neighbors. Thus, these instructions can be placed in a *compute* block. Recall that *compute* block is optional. Thus, these instructions can be also appear outside of the *olap-region*. There is no implicit barrier between the *olap-region* and the MPI_Allreduce operation. Thus, a task can start the *Allreduce* operation once it finishes the update without coordinating with other tasks. Likewise, after having *globalErr*, which is the maximum error among all tasks, a task can check the loop condition and go back to the mesh update phase if this error still larger than *threshold* or it hasn't reached *maxIters* iterations.

```
for (iter=0; iter<maxIters && globalErr>threshold; iter++)
{
    req_cnt=0;
    #pragma bamboo olap
    {
       #pragma bamboo receive
       {
          if(eastNeighbor) MPI_Irecv(..., eastNeighborID, ..., &req[req_cnt++]);
          ...west, north, south, up, ...
          if(downNeighbor) MPI_Irecv(..., downNeighborID, ..., &req[req_cnt++]);
       }
       #pragma bamboo send
       {
          if(eastNeighbor) MPI_Isend(..., eastNeighborID, ..., &req[req_cnt++]);
          ...west, north, south, up, ...
          if(downNeighbor) MPI_Isend(..., downNeighborID, ..., &req[req_cnt++]);
       }
       #pragma bamboo compute
       {
          MPI_Waitall(req_cnt, req);
          update (Uold, Un);
          swap (Uold, Un);
       }
    }
    if(iter % frequency==0){ //reduce error every "frequency" iterations
      err = Error(Uold);
      MPI_Allreduce(&err, &globalErr, ...);
    }
}
```

Figure 3.8. First example: a 3D Jacobi iterative solver. Each process exchanges ghost-cells with its 6 neighbors.

```
#pragma bamboo olap
{
 #pragma bamboo send
 {
   if(isLeaf(myRank))
     MPI_Send(sbuf, ..., parent, ...);
 }
 #pragma bamboo receive
 {
   for(child in myChildren){
     MPI_Recv(recvbuf+offset(child), ..., child, ...);
     reduce_op(recvbuf);
   }
   if(myRank!=root && isLeaf(myRank)==0)
     MPI_Send(recvbuf,..., parent, ...);
 }
}
```

Figure 3.9. Second example: an implementation of MPI_Reduce using the binomial tree algorithm. MPI_Send called by non-leaf nodes are grouped into a *receive* block with the MPI_Recv calls since the send operation must wait for the completion of all receive operations.

The second example is an implementation of the MPI_Reduce operation using a binomial tree-based algorithm as follows. The root node of the tree has log(P) children and no parent. Leaf nodes of the tree only have a single parent and no children. Other nodes are called internal nodes and have exactly one parent and log(P) children. In this algorithm, each left node sends data to its parent. Other nodes receive data from children and compute the local reduction operation. Except for the root node, processes send the interim results to the parent. Since MPI_Send called by leaf nodes are indepdent of other activites, we place it in a *send* block. The MPI_Send called by interal nodes has to wait for the result of MPI_Recv. Thus, we place these 2 calls in a *receive* block.

3.4 Task graph execution

Bamboo translates an annotated MPI program into a task graph. An instance of such a graph comprises a set of tasks executing and communicating in a different manner compared to the original MPI program. This section presents the Bamboo's execution and communication models, which explain the execution behaviors of tasks as well as the relation between the task execution and the communication handler and task scheduler.

3.4.1 Communication

Traditional communication models use handshake protocols to send messages from source to destination. For example, a communication transaction of the 3-phase handshake protocol begins with a *request-to-send* signal from the sender, followed by a *clear-to-send* signal from the receiver, and ends with a *data-transfer* event from the sender. During the communication, computations are blocked at the sender and receiver. Thus, the program execution suffers from latencies caused by *request-to-send* and *clear-to-send* signals and the time for *data-transfer* due to network bandwidth.

Bamboo, on the contrary, employs a non-imperative communication model by relying on communication handlers that can operate independently of task computations. Bamboo tasks communicate via messages, each corresponding to an edge of the task graph. Thus, we also call the communication handler that services the communication between tasks *message handler*. At the sender side, whenever an output is available, task wraps it up into a message so the *message handler* can send the data to the destination. The *message handler* maintains a buffer of messages. Thus, tasks do not have to wait for a response of the *message handler*. Rather, tasks can immediately continue the execution. At the receiver side, there is a *message handler* that listens for incoming messages. This *message handler* also works concurrently with task execution. Thus, it does not matter whether the receiving task is in the *WAIT* or *EXEC* state, or is executing when a message arrives. The *message handler* may buffer incoming messages, and tasks will consume buffered messages when they require inputs. Similar to MPI, messages will arrive at the destination in the same order that they are sent. At the task graph level, each message includes a fixed-size header [1] and a variable-length field for data. The header of each

[1]The message handler and lower layers of the network software stack may insert additional headers into the message that they service.

message consists of the information of *source* and *destination*, as well as a *sequence number*. *Sequence numbers* are simply integers that are used to distinguish messages between the same *source* and *destination*.

In addition to the tuple *(source, destination, sequence number)*, the message header may also contain performance meta-data that can be seen by the *message handler* for optimization purposes. For example, *aggregation* is a flag indicating that the *message handler* should bundle messages that have the same destination into a single message to save the startup cost of sending small messages. A more sophisticated example is that messages can have a flag indicating that they contain only the address of data. Communicating by addresses is an effective optimization on computer platforms that have a deep memory hierarchy such as GPU. On such platforms, tasks execute on the GPU but communicate via the host. If tasks residing in the same GPU communicate raw data, we will waste bandwidth of the PCIe connection between GPU and the host memory. Thus, a task can simply send a message containing the address of data, with the appropriate flag turned on so that the destination can interpret the message correctly.

3.4.2 Task scheduling

Based on the task dependency graph defined by Bamboo, the programmer can exploit task-grained parallelism by instantiating a graph with more tasks than *worker threads*. In the scenario of having a limited amount of resource (i.e. available *worker threads*) compared to the actual computing demand (i.e. *runnable* tasks), task scheduling plays an important role in realizing good performance. As a result, it is essential to construct an execution model that clearly describes how tasks can be scheduled and the task behavior during the execution of the graph.

Task schedulers

The task graph's runtime system employs a thread-pool model to execute tasks. In particular, within a single memory address space, the runtime system can be configured with one or multiple *worker threads* that run to completion. There is also a pool of *runnable* tasks. This task pool can be implemented as a single queue or be split into multiple queues, at the choice of the user. Each *worker thread* operates a scheduler, which dynamically selects tasks in a task queue to execute. While each *worker thread* is permanently bound to a processor, tasks can move among *worker threads* to balance computations among processors.

Message-driven execution model

Fig.3.10 presents the message-driven execution model employed by Bamboo. Unlike in the BSP model, where processes execute a sequence of communication and computation phases, Bamboo tasks do not explicitly wait on communication. Rather, a task runs until it meets the entrance of an *olap-region* (i.e. *evaluation point*), where it checks whether or not all inputs are ready to advance the execution. If some input is not available, the task returns control back to the scheduler corresponding to the processor on which the task is running. Upon a task return, the scheduler selects another runnable task to execute on the processor.

As previously mentioned, a task has an associated state, which is recognized by the scheduler. Task's state alternates between *EXEC*, *WAIT*, and an intermediate state called *firing rule evaluation* as shown in Fig. 3.11. In this state transition diagram, circles are states and arrows represent events. *Firing rule* can be considered as an object with internal data and a method that a task uses to make the decision on state transition. Data of the *firing rule* object includes arrived messages that have not been claimed by tasks. The *firing rule* method returns *true* if the corresponding task has received all required messages defined within the current *input window*, otherwise it returns *false*. The state transition is as follows. A task is *runnable* if and only if its state is *EXEC*. When a task executes and meets an *evaluation point*, it invokes the *firing rule* method. If this method returns *false*, task changes its state from *EXEC* to *WAIT*. Moreover, task has to return control back to the scheduler due to lack of input data. In response to an incoming message, a task that is the recipient of the message changes its state to *EXEC* if and only if the current state is *WAIT* and the firing rule returns *true*. Once a task is already in the *EXEC* state, it will not change state upon message arrivals. Rather, the *firing rule* object memorizes message arrival events in its internal data.

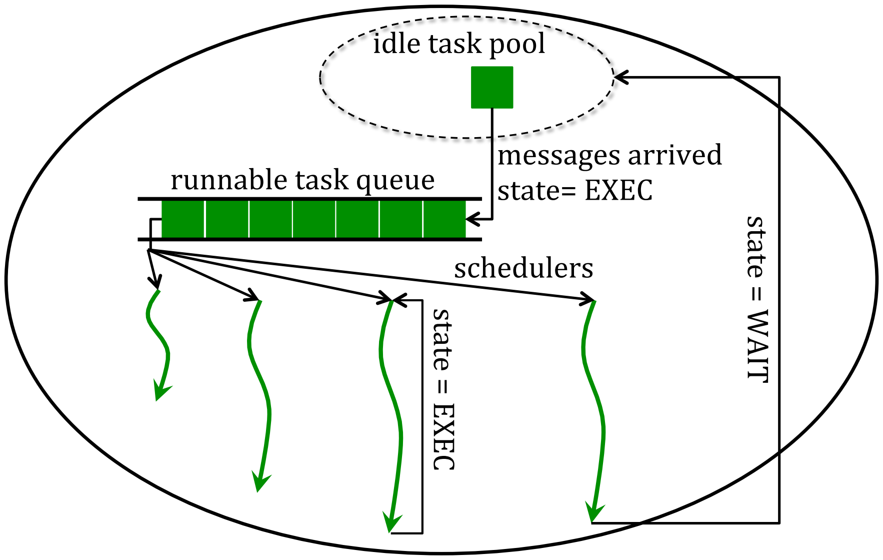


Figure 3.10. A message-driven execution model. Runnable tasks are scheduled to execute until they require more data from others. Tasks do not wait on communication but return control back to the schedulers. Upon message arrivals, idle tasks may become runnable again.

Non-preemptive, prioritizable task execution

The execution of Bamboo tasks is non-preemptive, meaning that the scheduler cannot suspend the execution of a task. Once a task is scheduled, it will run until its state changes to *WAIT* or the program has finished. This design choice was made in the

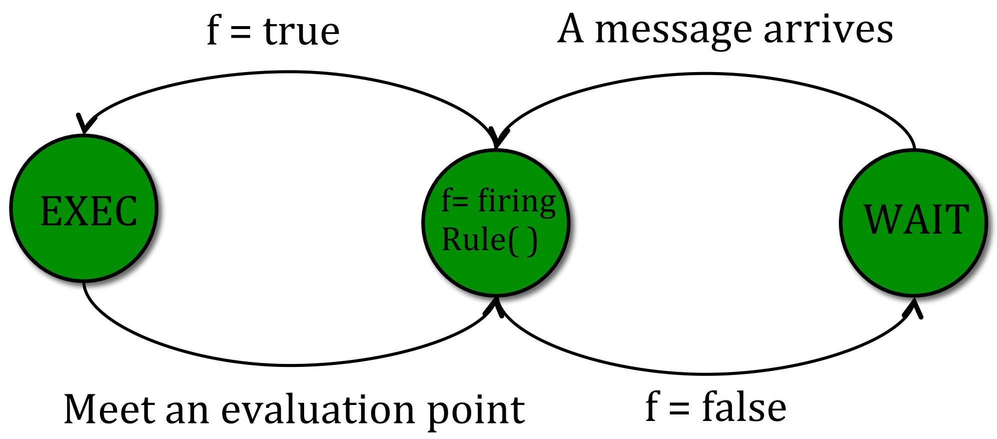


Figure 3.11. The transition between EXEC and WAIT states

interests of maintaining cache locality and reducing the number of task switches.

However, for irregular applications, where the amount of computation and communication significantly varies among tasks, a task prioritization model is needed to load balance the computation and overlap communication more efficiently. To this end, Bamboo provides the programmer with a scheme to prioritize tasks as follows. Each task has a priority that can be seen by the scheduler. The priority of a task is specified by the programmer and may change during the task execution. The parameters of a task scheduling algorithm now include both task state and task priority. Specifically, *runnable* tasks within a memory address space will line up in a priority queue. Among these tasks, the one with highest priority will be scheduled first.

To enhance the ratio of irregular communication that can be overlapped, Bamboo provides a task cooperation scheme. In this scheme, a task may cooperatively yield control even when it has enough data to continue the execution. The effect is to reduce the response time for tasks with high priorities, at the cost of possible increases in cache miss rate.

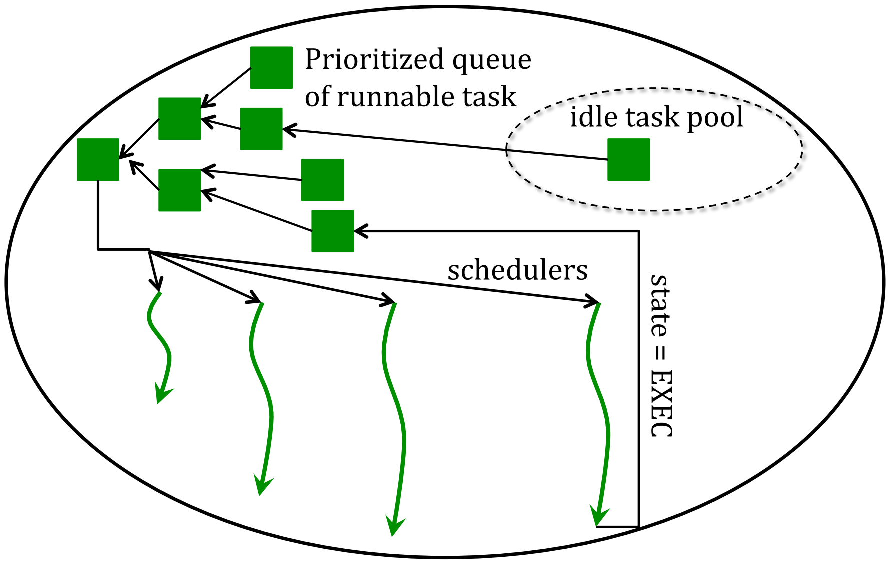


Figure 3.12. Among runnable tasks, the one with highest priority is scheduled. Tasks with negative priority return control even when they are runnable

3.5 Related work

Similar to Bamboo, Adaptive MPI (AMPI) [58] also applies the virtualization technique, but with a block-and-yield approach. In particular, multiple *virtual processors* (VPs) are spawned within an MPI process. Each VP is mapped into an OS thread. Whenever a running VP meets MPI_Recv, it blocks the execution and yields the physical processor to another VP. If data required by a VP has arrived, this VP can be rescheduled to continue its execution. The limitation of the block-and-yield technique is that the overhead of a large number of context switches may become a bottleneck. AMPI is implemented on top of Charm++ [60], which is built on top of MPI. Thus, AMPI can take advantage of the process migration capability supported by Charm++, thereby enabling dynamic load balancing. Like AMPI, Fine-grained MPI (FG-MPI) [59] applies the virtualization technique with a similar block-and-yield approach. However, FG-MPI integrates its runtime system into MPICH2, a popular MPI implementation. To enable faster context-switching and lower communication and synchronization overheads, the runtime system of FG-MPI employs a light-weight coroutine library, which provides more efficient yield for switching context between coroutines. FG-MPI, however, does not support process migration. By comparison, Bamboo does not employ the block-and-yield technique. Instead, Bamboo relies on a task dependency graph, where dependencies carry information of flows of data among tasks rather than an individual MPI call. In addition, instead of mapping a task to an OS thread, a Bamboo task is a user-defined thread. Thus, the task and data dependency representation of Bamboo allow a runtime system to schedule tasks more efficiently. Bamboo does not yet support task migration across processes. However, the prioritized task execution model in Bamboo supports dynamic load balancing to some extent.

Marjanovi et al. presented MPI/SMPSs [25], a hybrid model of MPI and dataflow that can overlap communication among MPI processes. In this hybrid model, the programmer taskifies MPI calls, specifying task inputs and outputs. As noted by the authors, taskifying communication may introduce potential deadlock. To resolve this issue, the scheduler forces the execution of tasks in the same order of creation, which may cause a negative effect in the application performance. HCMPI (Habanero-C MPI) employs a similar approach that requires the programmer to taskify MPI calls [26]. These two programming models do not employ any knowledge of MPI. By comparison, Bamboo relies on a translator to construct the task graph automatically, using domain specific knowledge of the MPI interface.

PLASMA [23] optimizes dense linear algebra applications such as LU factorization by dynamically scheduling a DAG (directed acyclic graph). DAGuE [61] is a framework that schedules a DAG on distributed memory. DPLASMA [24] is a distributed-memory implementation of linear algebra factorizations (Cholesky, LU, QR), and uses DAGuE. Although a DAG can be automatically translated from a serial code, efficiently distributing DAG tasks requires addition input from the user. DAGuEs user specifies a modifier, telling DAGuE how to transfer data from one task to another. By comparison, we utilize dependence information inherent to MPI code bases, where locality is already well considered. As a result, the programmer does not need to modify the translated code (which is often complicated) before executing it, though they do have to specify a modest amount of annotation.

3.6 Summary

1. Message Passing Interface (MPI) is a de facto standard for distributed-memory communication. Bulk Synchronous Parallel (BSP) is a programming model commonly used to develop MPI applications. However, programs written under the BSP model have to be highly optimized to deliver expected performance rates at large scales. For example, *split-phase* coding is a popular optimization technique that overlaps communication with computation. However, optimizing MPI code using *split-phase* coding is challenging since it requires non-trivial scheduling and communication policies to be embedded in the program, complicating the code and making it error-prone.

2. Bamboo factors scheduling issues out of the program execution by reformulating MPI code into a task dependency graph in which vertices are tasks containing computation code and edges are data dependencies between tasks. The graph maintains a partial ordering over the execution of tasks of the graph, enabling an external scheduler to run tasks based on the availability of data. In addition, Bamboo factors communication decisions out of the program execution by having a runtime communication handler to service the communication among tasks. To automate the translation from MPI source into the form of a task dependency graph, Bamboo employs a source-to-source translator. Though most of the transformation

phases are fully automatic, the translator requires a modest amount of user annotations in order to generate high quality code. Bamboo introduces a directive-based programming model so the programmer can quickly annotate MPI code.

3. Bamboo also introduces communication and execution models to describe task behavior at runtime. Tasks communicate via messages, serviced by *message handlers* that can run concurrently with the task. The execution of tasks is driven by messages. A task becomes *runnable* once all required messages have arrived. Tasks run until they need to wait for messages from others, at which time they return control back to the scheduler. Though task execution is non-preemptive, tasks can cooperatively yield control by setting a low priority.

3.7 Acknowledgements

This chapter, in part, is a reprint of the material as it appears in the article ”Bamboo - Translating MPI applications to a latency-tolerant, data-driven form” by Tan Nguyen, Pietro Cicotti, Eric Bylaska, Dan Quinlan and Scott Baden, which appears in the Proceedings of the 2012 ACM/IEEE conference on Supercomputing (SC’12), Salt Lake City, UT, Nov. 10 -16, 2012.

Chapter 4

Design and implementation

4.1 Software stack overview

Bamboo relies on a source-to-source translator to automatically transform an annotated MPI program into an equivalent graph-based, latency-tolerant form presented in Chapter 3. Although MPI offers hundreds of routines, a small subset of them can be used to implement the rest. For the sake of portability, we split the Bamboo translator into 2 software layers as shown in Fig. 4.1. The bottom layer handles a minimal set of MPI point-to-point primitives. We call this layer *core message passing*. An implementation of this layer highly depends on a runtime system that executes the generated graph program. Since the number of MPI routines supported by the *core message passing* layer is small (fewer than 10), porting this layer to similar runtime systems should not pose a big challenge. While the *core message passing* layer supports only basic primitives, the top layer of the Bamboo software stack supports the translation of complete MPI programs, which are commonly seen in practice. As a result, this layer supports a substantially richer set of MPI routines, including communicator splitting and collective operations. For that reason, we call it *utility* layer. This top layer divides into a few sublayers. Unlike the *core message passing* layer, the upper layers are independent of the underlying runtime system.

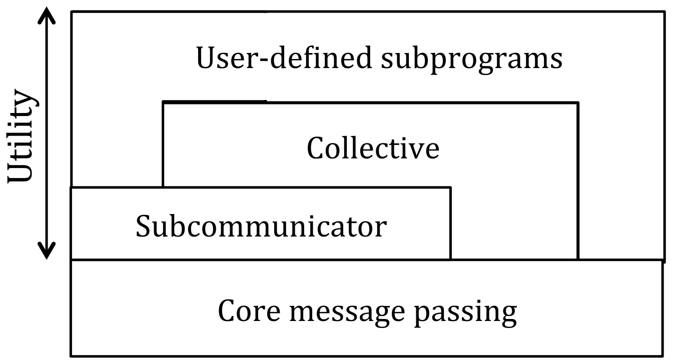


Figure 4.1. The software stack of the Bamboo source-to-source translator. The *core message passing* layer employs services provided by a runtime system. On top of this layer are implementations of MPI high-level routines, which are independent of the runtime system.

Currently the *core message passing* layer of the Bamboo software stack relies on services provided by Tarragon [52–54], a runtime system designed and developed by Pietro Cicotti. The reason for choosing Tarragon is that this runtime system has strongly demonstrated its ability to overlap communication with computation via scheduling a task dependency graph [52–54, 62, 63]. A Tarragon program generated by Bamboo comprises codes to define, construct, launch, and destruct a task dependency graph. However, this chapter will discuss only the code that defines the graph behavior, which must be derived from a specific application. The codes for constructing, launching, and destructing a task graph will not be presented as they are fairly simple and generic.

We employ the ROSE compiler framework [64] to implement the source-to-source translator. This framework comprises a*frontend*, a *middle-end*, and a *backend* as shown in the first column of Fig. 4.2. The *frontend* parses standard C source and generates an Intermediate Representation (IR), which is an in-memory Abstract Syntax Tree (AST). The *middle-end* translates the program via modifying the IR. Finally, the *back-end* completes the translation process by converting the IR back to source code.

Fundamental modules of Bamboo are built on top of the *middle-end* of the ROSE compiler framework. Bamboo also includes an auxiliary module called *MPI_Extractor*, sitting between the *frontend* and *middle-end* to extract information about the parameters passed to MPI functions, since the frontend considers the MPI calls as ordinary C function calls. The second and third columns of Fig. 4.2 describe fundamental modules of Bamboo, which are as follows. The *annotation handler* extracts information from each Bamboo directive along with the corresponding location within the IR. The *analyzer* and *transformer* modify the IR to reinterpret the program behavior. The *optimizer* applies various transformations to improve the quality of the generated source code.

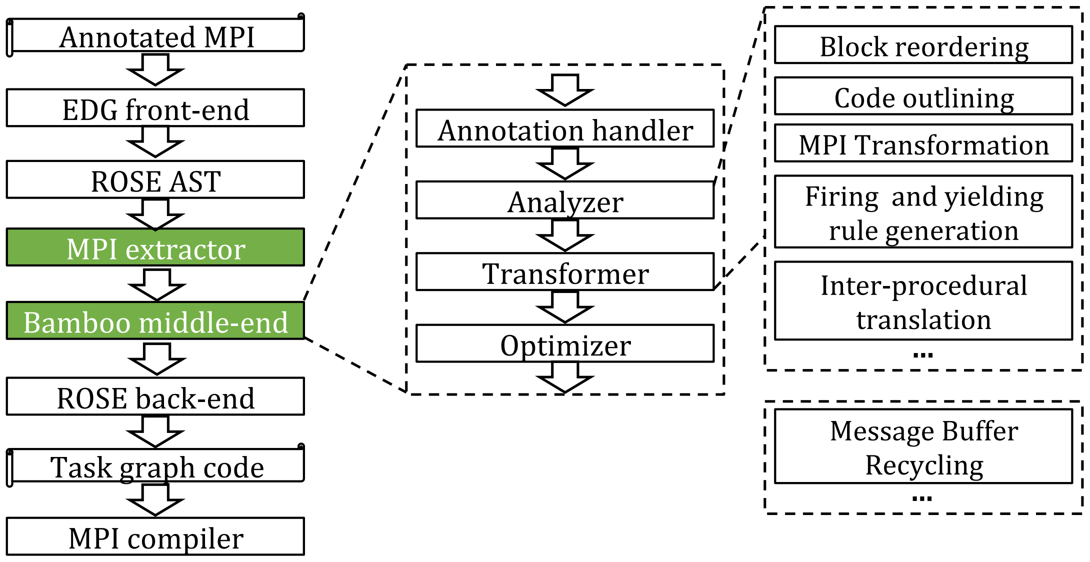


Figure 4.2. Bamboo is built on top of the ROSE compiler framework, which includes *frontend*, *middle-end*, and *backend*. Bamboo focuses in the *middle-end*, transforming and optimizing an intermediate representation generated by the *frontend*.

The remainder of this chapter presents implementation details of the Bamboo translator, starting with a brief overview of the Tarragon runtime system.

4.2 Tarragon in a nutshell

Tarragon is a library for implementing distributed-memory applications using an explicit task-dependency graph. The library includes an interface for developing the graph program and a run-time system for operating its execution. This section briefly presents the abstract model and implementation of Tarragon, as well as a few extensions to the runtime system investigated in this dissertation.

4.2.1 Tarragon interface

The Tarragon programmer explicitly constructs a task-dependency graph using an abstract interface provided by Tarragon, including four basic classes: *Graph*, *Task*, *Dependency*, and *Map*. *Graph* is a container of user-defined *Tasks*, which encapsulates application-dependent code. *Dependency* defines data dependencies between *Tasks* (AKA *edges* of the graph). The programmer explicitly connects *Tasks*, and Tarragon will automatically create *Dependency* objects for these connections. *Map* defines a multi-dimensional coordinate system, representing the *Task* topology.

The behavior of *Tasks* is defined by the programmer via 3 methods: *vinit()*, *vexecute()*, and *vinject()*. *Vinit()* initializes task data, and this method runs only for once at the beginning of the graph program. *Vexecute()* defines computation and may produce outputs for other tasks via messages. Tarragon pre-defines *put()*, a method to produce output. *Put()* is the sole data motion primitive defined by the Tarragon Task interface. This method will notify the Tarragon runtime system that a message is ready to be sent. *Put()* is asynchronous and hence a task can immediately continue its execution without waiting for a response from the runtime system. *Vinject()* is a lightweight method that defines a task's response on the arrival of messages. Although the programmer defines 3 methods *vinit()*, *vexecute()*, and *vinject()* of *Task*, they do not explicitly invoke these methods. Rather, it is the runtime system that executes tasks (by invoking *vinit()* and *vexecute()*) and handles communication among tasks (by invoking *vinject()*). We next present the implementation of the runtime system and the interaction between tasks and components of the runtime system.

4.2.2 Tarragon runtime system

Tarragon employs a distributed-memory runtime system for task graph program execution. The Tarragon runtime system comprises a task scheduler and a communication handler, built on a threading library (Pthreads) and communication library (MPI), respectively. Specifically, the runtime system is a distributed-memory program comprising multiple processes, each split into multiple threads. The main thread instantiates graphs and works as a communication handler. Thus, we also call this thread a *message handler thread*. Other threads operate as *worker threads*. *Message handler threads* communicate with each other using MPI send and receive primitives. *Worker threads*, however, do not communicate. Instead, these threads put messages into a buffer, which will be consumed by the *message handler thread*. Since there may be multiple *worker threads* within a process, the message buffer needs to be locked before a *worker thread* can safely insert a message. Both *message handler* and *worker* threads run to completion.

Shown in Fig. 4.3 is the execution of a task graph program, beginning with the initialization of the Tarragon runtime system. The runtime system then instantiates tasks of the graph. These tasks are executed by *worker threads*. Messages among tasks are handled by the *message handler thread*. Tasks cannot occupy *worker threads* when waiting for messages. Rather, tasks yield control when they require data from others. The communication handler of the runtime system delivers messages from source to destination. Upon the arrival of a message, the runtime system invokes *vinject()* of the receiver. This method determines whether the task has received all the messages necessary to become *runnable*. The interaction between tasks and the runtime system components (i.e. *communication handler* and *scheduler*) is via *state*. Specifically, each task maintains a state that can be seen by the runtime system. The task state can be recognized by the scheduler of the runtime system, enabling the scheduler to invoke

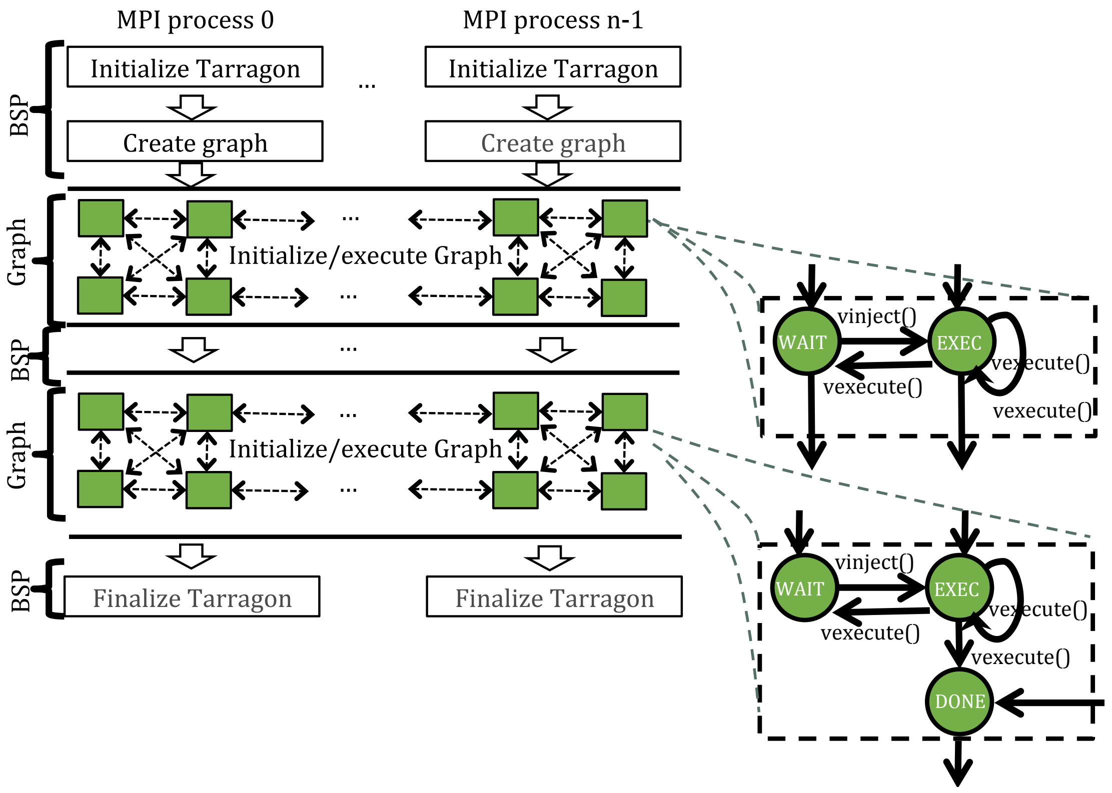


Figure 4.3. A sketch of a task graph execution. Graph construction and teardown work in BSP mode whereas tasks of the graph execute in data-driven mode

vexecute() when the state is *runnable*. After executing *vexecute()*, if the task needs to wait for data from other tasks, it changes the state to *wait* and yields control. The scheduler will swap in another task and invoke the *vexecute()* method of the new task. Each *worker thread* maintains a task scheduler, dequeuing one task at a time from a queue of *runnable* tasks. To support task priority, each task queue can be configured as a priority queue. *Worker threads* can only see task meta-data, including task state and priority. These threads cannot write to task's data, nor suspend the execution of a task. Once a task finishes executing, it exits the graph mode and return control to the runtime system to launch another graph or to finalize the application.

4.2.3 Extensions to the runtime system

To obtain specific optimizations required by a few classes of applications and platforms, this dissertation proposes a few extensions to the design and implementation of the Tarragon runtime system as follows.

First, Tarragon was originally built for SMP (Symmetric Multiprocessing) clusters. Thus, the Tarragon communication handler can process messages stored in processor memory only. For applications that offload compute-intensive work and corresponding data to accelerators such as Graphics Processing Units (GPUs), the occupying task suspends its execution when transferring data between host memory (i.e. processor memory) and device memory, preventing the resulting delay from being hidden by any computations on the host and device. Thus, we extended the communication handler so that data transfer between host and device is transparent to the programmer and can be overlapped with computation. In particular, the incoming and outgoing message buffers can now store data in device memory. We next extended the communication handler to handle data transfer between host and device. In addition, if a message is to be transferred to the device memory, the runtime system will not invoke the *vinject()* method until this data transfer completes. This extension factors host-device transfer out of the task execution. Thus, it allows task computations (on host, device, or both) to be overlapped with data transfer between host and device.

Second, the Tarragon scheduler was originally designed to keep tasks executing as long as they are *runnable*. Even when a task is willing to relinquish control in the middle of its execution (which can be done by unconditionally jumping over the last instruction of the *vexecute()* method), it will be immediately rescheduled by the scheduler. In certain situations, tasks should work more cooperatively by yielding control to other tasks. To this end, we extended the Tarragon scheduler to allow a task to return control at the time of its choosing. In particular, the value of task priority is divided into positive and negative domains. We modified the scheduler so that when it observes an occupying task leaving with a negative priority, it will put the task back to the task queue and select another task with the highest priority to execute. With this scheme, a task can relinquish control at any time by setting its priority to a negative value, save the latest instruction, and return from the *vexecute()* method.

4.3 Translation: core message passing layer

We translate MPI applications into the form of a task dependency graph operating under the control of the Tarragon runtime system. MPI consists of a small set of fundamental point-to-point primitives and a predefined process set called *MPI_COMM_WORLD* communicator. Based on this minimal set of MPI primitives, the MPI programmer can implement any application. The *core message passing* layer of the Bamboo translator reformulates an MPI application programmed with these primitives into its equivalent task dependency graph. In particular, the *core message passing* layer supports rank and size queries (*MPI_Comm_rank* and *MPI_Comm_size*), point-to-point communication routines (*MPI_Send*, *MPI_Isend*, *MPI_Recv*, *MPI_Irecv*, *MPI_Wait*, and *MPI_Waitall*), and *MPI_COMM_WORLD* communicator. In this section, we present the implementation of this layer, relying on the Tarragon runtime system as shown in Fig. 4.4. The implementation of the *Bamboo utility layer*, which is independent of Tarragon, will be presented in the next section.

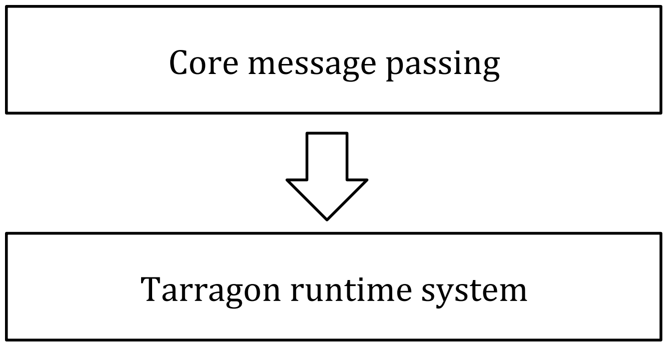


Figure 4.4. The core message passing layer queries services provided by Tarragon runtime system

4.3.1 Block reordering

As presented in Chapter 3, the input source of the Bamboo translator is annotated with *olap-regions*, each consisting of communication blocks classified into *send* and *receive* blocks, which further contain MPI function calls. Due to the differences between MPI and Bamboo's interpretation, Bamboo performs a simple code transformation called *block reordering*. In particular, Bamboo will reorder certain *communication blocks* in certain situations. For example, the left side of Tab. 4.1 shows a common communication pattern used in MPI applications that will be restructured by Bamboo. Specifically, Sends (in the *send block*) issued by a process match up with receives (in the *receive block*) of the other process encoded in the same iteration. Bamboo has to reorder the *send block* due to the following reason. A task is *runnable* only when all necessary data is available. If we place the corresponding send within the same iteration as the corresponding receive, data sent in one iteration will not be received until the next. But, the algorithm needs to receive data within the same iteration. To cope with this timing problem, Bamboo reorders the send block, advancing it in time so that the sending and receiving activities reside in different iterations. Bamboo will set up a pipeline, replicating the send block to the front of, and outside, the iteration loop. It also migrates the existing call to the end of the loop body, adding an appropriate guard derived from the loop iteration control logic. After reordering, the transformed code appears as shown in the right side of Tab. 4.1. The send and receive blocks now reside in different iterations, preserving the meaning of the original code.

Table 4.1. Left: a typical MPI program that requires code reordering. Sends within the send block of a process match with receives within the receive block of another process in the same iteration. Right: The same code with send reordered.

Before reordering	After reordering

`#pragma bamboo olap`
`for (i=1;i<=nIters;i++){`
`  Receive block`
`  Send block`
`  Compute block`
`}`
	`i=1`
	`if(i<=niters) Send block`
	`#pragma bamboo olap`
	`for (;i<=nIters;){`
	`  Receive block`
	`  Compute block`
	`  i++`
	`  if(i<=niters)`
	`    Send block`
	`}`

4.3.2 Code outlining

As described in Sec. 4.2, Tarragon provides an abstract class called *Task*. Bamboo derives a concrete *Task* class from this abstract class, implementing the three methods of *Task*: *vinit()*, *vexecute()*, and *vinject()*. *Code outlining* is the technique that Bamboo uses to generate code for these methods. Lines 1 to 10 of Fig. 4.5 show a code snippet containing an *olap-region* in a universal form where both loop and conditional statement (in the loop) present.

Olap-regions that contain either loop or conditional statement, or none of them can be treated as special cases of this form. The code outlining is as follows. Code prior to the first *olap-region* is simply outlined to *vinit()* without being modified. The remainder, including *olap-regions* and unannotated codes, is outlined to *vexecute()* as shown by lines 14 to 49 of Fig. 4.5. *Receive blocks* in *olap-regions* are also processed further to implement *vinject()*, which will be presented later on. Each *olap-region* will be split into *olap-init* and *olap-body* portions. *Olap-init* contains *send blocks* of the first iteration of the loop, which enable the input for the first iteration of the current *olap-region*. Since *send blocks* in *olap-init* of an *olap-region* correspond to outputs of the prior *olap-region*, it is not neccessary to check if the data is available to be sent as they are definitely ready. The *olap-body* code, however, requires all inputs to be available before it can execute. Note that before a task executes the *olap-body* code, it returns control back to the scheduler whether or not all inputs have been received. If some inputs are not ready, the task changes its state to *WAIT* before returning, and the scheduler will pick another task to execute. If all inputs are ready, task state will remain *EXEC*, and the scheduler will immediately reschedule it. Although in the latter case we may waste a few clock cycles for 2 unconditional jump instructions and a method invocation, the scheduler can obtain important information about the task execution, e.g. information that can benefit performance profiling and enable smart scheduling algorithms for load balancing.

4.3.3 MPI transformation

Once the program has been outlined to *vinit()* and *vexecute()*, MPI routines are either translated directly into Tarragon equivalent routines, or analyzed for valuable information, or simply removed.

```
prior codes
#pragma bamboo olap
for(int iter=0; iter<maxIters; iter++){
  #pragma bamboo send
  {}
  #pragma bamboo receive
  {}
  #pragma bamboo compute{}
}
successive codes

―――――――――――――――――――――――――――――――――――――――

vexecute(){
  goto olapInit/Body...; //either Init or Body
  ...
  prior codes
  olapInitN:
    iter = 0;
    if (iter<maxIters)
    {
       #pragma bamboo send
       {
       }
       if(inputReady==false) state= WAIT;
       goto exit;
    }
  olapBodyN:
    if (iter<maxIters)
    {
       #pragma bamboo receive
       {
       }
       #pragma bamboo compute
       {
       }
       iter++;
       if (iter<maxIters)
       {
         #pragma bamboo send
         {
         }
         if(inputReady==false) state = WAIT;
       }
       goto exit;
    }else {updateCurrentOlap();}
    successive codes...
  exit: //if state is EXEC, task will be rescheduled immediately
}
```

Figure 4.5. Bamboo outlines an annotated MPI program into vexecute(). Before program execution can enter the body of an *olap-region*, the status of all inputs corresponding to the *olap-region* is checked to determine the task state.

MPI_comm_ rank and MPI_Comm_size

The *core message passing* layer supports the MPI_COMM_WORLD communicator. Thus MPI_comm_ rank and MPI_Comm_size routines return the rank of a process and the total number of MPI processes in this communicator, respectively. Under the form of a task dependency graph, we also have *task ID* and graph size. Since the virtualization from MPI process to task does not change the communication pattern, Bamboo simply rewrites the calls to MPI_Comm_rank() and MPI_ Comm_size() to corresponding method invocations that return the task ID and number of tasks in the graph, respectively.

MPI point-to-point communication primitives

Bamboo currently supports the following 6 point-to-point communication primitives: MPI_Send, MPI_Isend, MPI_Recv, MPI_Irecv, MPI_Wait, and MPI_Waitall. The translation of these functions is as follows.

- MPI_Send and MPI_Isend (*send*): each *send* is translated to a task output. In particular, Bamboo creates a message and fills the message header with information that can help the Tarragon communication handler deliver the output data from the task to the destination. Then Bamboo copies communicated data from the outgoing buffer of *send* into the data buffer of the message. Finally, Bamboo generates an invocation of *put()*, notifying the communication handler of the Tarragon runtime system that the output data is ready to be shipped.

- MPI_Recv and MPI_Irecv (*recv*): Tasks do not explicitly invoke any method to receive data from other tasks. Instead, the *message handler thread* receives and buffers incoming messages. When a destination task stops its execution, the *worker thread* will invoke *vinject()* of the task to hand over the message to task. This method also determines whether the task can change its state from *WAIT* to *EXEC*. Bamboo uses all *recv* calls within an *olap-region*, together with any conditional statements connected with them, to implement *vinject()*. Details of *vinject()* implementation will be discussed in Sec. 4.3.4. It's worth noticing that *vinject()* can be invoked by the runtime system even when task state is *EXEC* but the task has not been scheduled to execute. In this case the task will remain at the *EXEC* state. However, it may happen that a task over-receives data that are required by multiple executions of *vexecute()*. Thus, Bamboo has to generate code so that each task can keep track of these messages for later uses. In addition, when a task is about to leave *vexecute()* after one execution, it only changes state to *WAIT* if input data for the next execution is not ready, as shown in the code inlining. If tasks fail to do so, they will be idle forever since vinject() was already invoked when task over-received messages and thus can't be invoked again. Details of how Bamboo handles this case will be also discussed in Sec. 4.3.4.

- MPI_Wait and MPI_Waitall: Recall that each *olap-region* requires all inputs to be available before it can execute. Due to this characteristic, waiting at MPI_Wait and MPI_Waitall is no longer necessary. As a result, Bamboo simply removes these function calls.

4.3.4 Firing rule and yielding rule

During execution, the task state cycles between *WAIT* and *EXEC*. Whereas the task state can change from *EXEC* into *WAIT* via *vexecute()*, it can move from *WAIT* to *EXEC* via *vinject()*. The condition that determines when *vinject()* can enable a transition from *WAIT* to *EXEC* is called the *firing rule*. The formula that *vexecute()* uses to reverse the state transition from *EXEC* into *WAIT* is called the *yielding rule*. Bamboo extracts information from MPI receive calls and associated conditional statements to generate *firing* and *yielding rules*.

Let m and C be, respectively, a message possibly received by a process in an *olap-region* and the associated conditional statements. Whether a particular process should wait for message m or not is subject to the evaluation of the condition C. Thus, the *firing rule* for an *olap-region* can be written in the conjunctive normal form.

$$\bigwedge(\neg C_i \bigvee m_i) \tag{4.1}$$

The *yielding rule*, on the contrary, can be presented in the disjunctive normal form.

$$\bigvee(C_i \bigwedge \neg m_i) \tag{4.2}$$

Where i ranges from 1 to the number of messages possibly received by a process in an *olap-region* and m_i is true means that message i has arrived.

Lines 1 to 12 in Fig. 4.6 present an MPI example containing an *olap-region* where each process receives messages from multiple processes. The implementation of the *firing rule* is shown at lines 15 to 34 in Fig. 4.6, where a task changes its state into *EXEC* if and only if all required messages are available. This requires the communication handler of the runtime system to invoke vinject() every time a message arrives. Bamboo reuses the code from line 17 to line 27 for implementing the *yielding rule*, as a task can run this code for only once to determine whether or not it needs to yield control to another task.

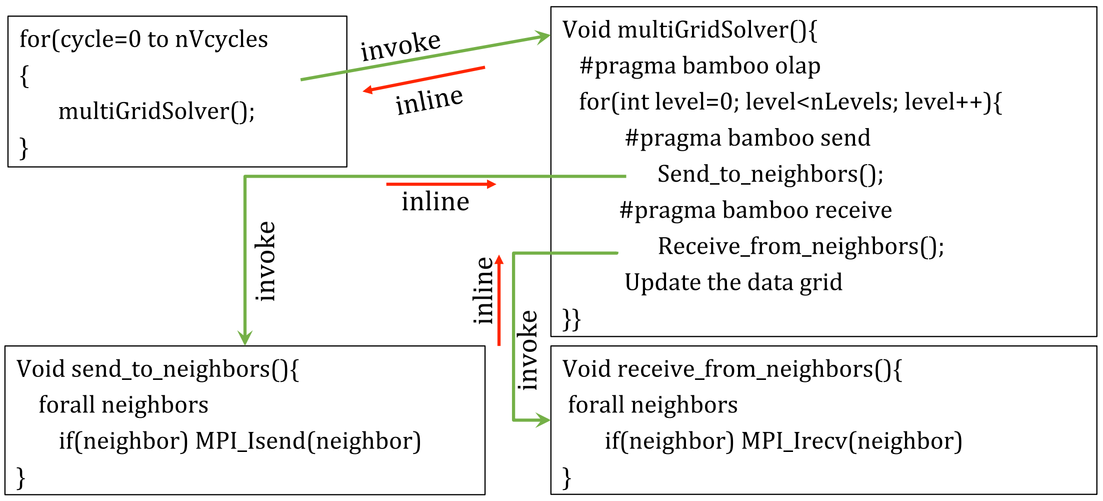
4.3.5 Interprocedural translation

The code transformation and analysis modules of Bamboo may need to run across procedures. For instance, Fig. 4.7 presents an example where the source codes of *olap-region* and *communication blocks* (i.e. *send block* and *receive block*) reside in different

```
1  #pragma bamboo olap
2  {
3    #pragma bamboo receive
4    {
5      for(int i=0; i<numPartners; i++){
6       if(condition(i))
7         MPI_Recv(..., i, ...);
8      }
9    }
10   #pragma bamboo send
11    ...
12 }
13
14 ————————————————
15 void task::vinject(message msg){
16  messsage.push(msg.source, msg);
17  bool fireable = true;
18  for(int i=0; i<numPartners; i++){
19   if(condition(i))
20    if (message[i] == NULL){
21      fireable = false;
22      goto exit;
23    }
24  }
25  exit:
26  if(fireable) state = EXEC;
27  else state = WAIT;
28 }
29
30 while(true){ //run by the communication handler
31   listen for a message
32   int dest = message.destination;
33   taskGraph[dest].vinject(message);
34 }
```

Figure 4.6. Lines 1-12: an MPI code annotated with *olap-region*. Lines 15-34: the firing rule code for the *olap-region*. The *yielding rule* can be obtained by simply reusing *vinject()* code.

procedures. To generate *firing* and *yielding rules* for the *olap-region*, the translator needs information in the *receive block*. Inlining is a technique that exposes the calling context to the procedure's body and the procedure's side effect on the caller. Bamboo performs inlining, and the process is as follows. If a procedure other than main() directly or indirectly invokes MPI calls, Bamboo registers it as an *MPI-invoking procedure*. Bamboo will subsequently inline all *MPI-invoking procedures* from the lowest to the highest calling levels. The inlining process is transparent to the programmer and does not require any annotation. Moreover, since Bamboo inlines *MPI-invoking procedures* only, the amount of code requiring inlining is small. Our inlining strategy, however, currently does not support recursive procedures.



```
for(cycle=0 to nVcycles
{
    multiGridSolver();
}
```

```
Void multiGridSolver(){
  #pragma bamboo olap
  for(int level=0; level<nLevels; level++){
      #pragma bamboo send
          Send_to_neighbors();
      #pragma bamboo receive
          Receive_from_neighbors();
      Update the data grid
}}
```

```
Void send_to_neighbors(){
   forall neighbors
      if(neighbor) MPI_Isend(neighbor)
}
```

```
Void receive_from_neighbors(){
 forall neighbors
     if(neighbor) MPI_Irecv(neighbor)
}
```

Figure 4.7. A multigrid solver with call chains containing MPI invocations. Bamboo registers procedures that directly or indirectly invoke MPI calls as an *MPI-invoking procedure*. It then inlines all *MPI-invoking procedures* from the lowest to the highest calling levels.

4.3.6 Message buffer recycling

Under the data-driven model that Bamboo supports, tasks do not directly communicate with each other. Instead, they produce output for other tasks via messages, which will be delivered by the communication handler of the runtime system. The overhead of allocating and deallocating memory for messages is significant, especially when tasks communicate with many small messages. To avoid such overheads, the communication handler of the Tarragon runtime system and tasks of the graph maintain their own message pool. The memory buffer allocated for each message is fixed, enabling it to be reused for data input/output of any size. For each incoming input, the communication handler extracts a message from its message pool to buffer the data. This message will be passed to the destination task. After this task reads the data of the message, it will not delete the message but keep it in the message pool for later use as follows. When the task produces output for another task, it will extract the message from the message pool and copy data to the message buffer. This message will be passed to the communication handler via the *put()* method. Thus, we can see that tasks and the communication handler maintain a circulating message system, where after a certain point of the program execution no further message creation is required. The overheads of allocating initial messages will be amortized by the total execution time of the program.

With the current implementation of the message buffer recycling mechanism, the buffer size is fixed and must be at least equal to the size of the largest message used in the program. Thus, if at some phase in the application, tasks communicate on many small messages with high degree of parallelism, we may run out of memory. For example, in the LU factorization algorithm the largest message can be 512MB or even larger. If we run the corresponding task graph with 2^{20} tasks, each task needs to buffer $log_2^{2^{20}}$*512MB = 10GB when the program performs a barrier operation. This memory demand is not affordable as a compute node may contain many tasks. An efficient remedy for this problem is that when the size of a message is larger than a pre-defined threshold, we split it into multiple chunks. The threshold can be significant so that the startup costs are negligible. Currently this capability is not yet supported by Bamboo, since we haven't employed that many tasks. However, we will implement this approach in a near future.

4.4 Translation: utility layer

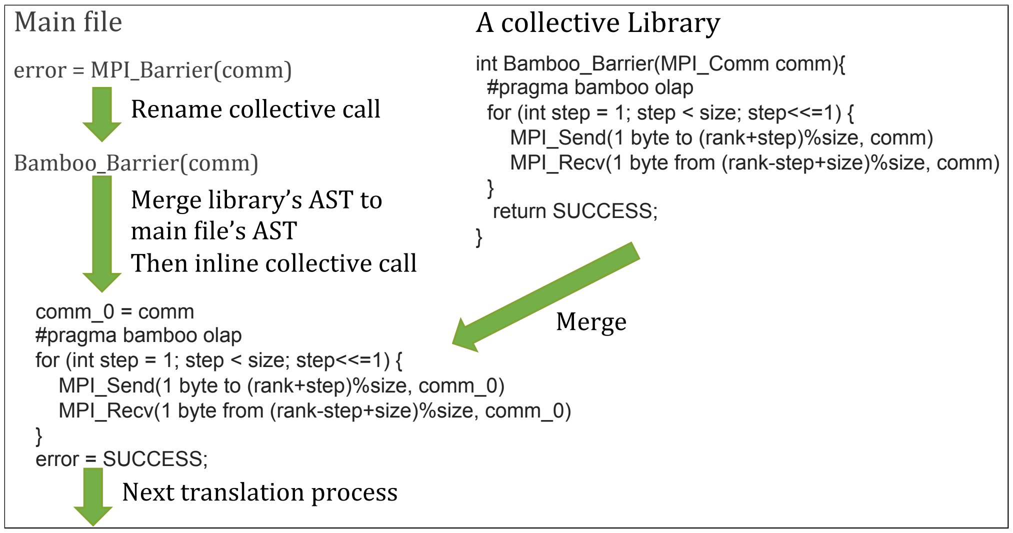
On top of the *core message passing* substrate, hundreds of high-level routines can be built. These routines include collective functions and communicator support. This section presents a library-based approach that allows system vendors and MPI programmers to easily translate a custom implementation of these routines into a task graph form. Bamboo also includes its own implementation of commonly used routines.

4.4.1 Collective

While MPI specifies the interface of collective operations, it does not insist on a particular implementation of these functions. Thus, the programmer is free to implement the collectives according to the needs of the hardware and the application. Bamboo also includes its own implementation of collective functions. In particular, Bamboo maintains a library implementing widely used collective functions, by breaking them down into their point-to-point components. The source-to-source translator will automatically detect non-point-to-point MPI function calls in the MPI input source and inline corresponding implementations into the program's source code before translating these codes together into a task graph form.

Bamboo employs the AST merge mechanism provided by the ROSE compiler framework as shown in Fig. 4.8. This merge mechanism allows the ASTs generated from source codes in different files to be merged into a single AST. In order to avoid any possible conflict between MPI and Bamboo libraries, all routines in the Bamboo library start with the prefix *Bamboo* instead of *MPI*. For example, the *MPI_Barrier()* routine in the MPI collective interface corresponds to the *Bamboo_Barrier()* implementation in the Bamboo library. Collective calls in the MPI program, however, do not have to be modified.

Bamboo automatically renames the MPI prefix of collective calls (e.g. MPI_Allreduce to Bamboo_Allreduce), then it inlines the code of Bamboo-prefixed functions into the application. We call this mechanism *plug-and-translate* since the programmer can easily incorporate a custom implementation of collective operation into the library and translate it with the application.



```
int Bamboo_Barrier(MPI_Comm comm){
  #pragma bamboo olap
  for (int step = 1; step < size; step<<=1) {
     MPI_Send(1 byte to (rank+step)%size, comm)
     MPI_Recv(1 byte from (rank-step+size)%size, comm)
  }
  return SUCCESS;
}
```

```
comm_0 = comm
#pragma bamboo olap
for (int step = 1; step < size; step<<=1) {
   MPI_Send(1 byte to (rank+step)%size, comm_0)
   MPI_Recv(1 byte from (rank-step+size)%size, comm_0)
}
error = SUCCESS;
```

Figure 4.8. The multi-file translation framework using the AST merge mechanism

Fig. 4.9 shows an implementation of the MPI_Scatter operation. Similar to the implementation of MPI_Reduce previously shown in Chapter 3, the MPI_Scatter operation employs the *Binomial Tree* algorithm. For the scatter case, each process has a parent and log(P) children. The root distributes portions of the scatter buffer to its children (starting with the smallest child), including extra data that it requests the children to forward deeply downward to the leaves of the process tree. During the time that a parent is scattering data at the level of its children, these children also forward parts of the data to their children. The scattering process continues until all leaves of the *Binomial Tree* have received data. Since the width and the height of the *Binomial Tree* are both log(P), the cost of MPI_Scatter using this algorithm is log(P). Tab. 4.2 shows algorithms that Bamboo uses to implement common collective operations and the corresponding latency and bandwidth costs.

```
int MPI_Scatter(void *sbuf, int scnt, MPI_Datatype stype, void *rbuf, int rcnt,
    MPI_Datatype rtype, int root, MPI_Comm comm)
  int myRank, commSize;
  MPI_Comm_rank(comm, &myRank);
  MPI_Comm_size(comm, &commSize);
  char* tempbuf;
  if(myRank==root) tempbuf= (char*)sbuf;
  else tempbuf= (char*)malloc(scnt*commSize*sizeof(stype)/2);
  #pragma bamboo olap
  {
   #pragma bamboo send
   {
     if(myRank==root)
       for(child in myChildren)
         MPI_Send(sbuf+offset(child),count,dtype,child,tag,comm)
   }
   #pragma bamboo receive
   {
     MPI_Recv(tempbuf+recvOffset,count,dtype,parent,tag,comm,stt)
     if(myRank!=root && isLeaf(myRank)==0)
       for(child in myChildren)
         MPI_Send(tempbuf+offset(child),count,dtype,child,tag,comm)
   }
  }
 }
 memcpy(rbuf, tempbuf, rcnt*sizeof(rtype));
```

Figure 4.9. An implementation of MPI_Scatter using the Binomial tree algorithm.

Since Bamboo break collectives into their point-to-point components, the communication delays incurred by these operations can be overlapped with available computation. However, the costs may increase as the number of tasks increases. Fig. 4.10 shows the performance of MPI_Scatter as we increase the number of tasks in the absence of useful computation. It can be seen that with a small scatter size, the communication time is dominated by startup costs. In this scenario, using more tasks will increase the communication costs. However, we can see that as the scatter size grows the communication time is dependent on the available bandwidth. Thus, we can increase the virtualization factor without harming the performance as the total volume of data is independent of the number of tasks.

Table 4.2. Default implementation algorithms of collective operations. The $\alpha\beta$ cost model is used to estimate the cost of collective operations, where α is latency and β is inverse bandwidth [5].

Collective API	Algorithm	Complexity
MPI_Barrier	Bruck's algorithm	$\lceil lgP \rceil \alpha$
MPI_Bcast	Binomial Tree	$\lceil lgP \rceil (\alpha + \text{s}\beta)$
MPI_Reduce	Binomial Tree	$\lceil lgP \rceil (\alpha + \text{s}\beta + \text{size*opCost})$
MPI_Allreduce	Recursive doubling	$\lceil lgP \rceil (\alpha + \text{s}\beta + \text{size*opCost})$
MPI_Scatter	Binomial Tree	$\lceil lgP \rceil \alpha + \text{totalSize*}\beta$
MPI_Gather	Binomial Tree	$\lceil lgP \rceil \alpha + \text{totalSize*}\beta$
MPI_Allgather	Bruck's algorithm	$\lceil lgP \rceil \alpha + \text{totalSize*}\beta$
MPI_Alltoall	Bruck's algorithm	$\lceil lgP \rceil (\alpha + \frac{s}{2}\beta)$

4.4.2 Communicator

An MPI Communicator is a namespace describing the set of MPI processes that each process can communicate with for a particular MPI routine. The set of these processes is an ordered list, ranging from 0 to P-1, where P is the number of processes in the set. *MPI_COMM_WORLD* is the only predefined communicator in the MPI environment, defining the order of all processes of an MPI program. Based on this predefined communicator, MPI allows the programmer to derive new communicators to realizing the following capabilities.

First, the programmer splits the process set into smaller subsets, which can be further partitioned. For this purpose, communicator partitioning is effectively a means of simplifying the code design rather than improving performance. For example, performing a broadcast operation within a process row (or column) of a 2D process grid is commonly seen in practice. The programmer can partition MPI processes into rows (or columns) so

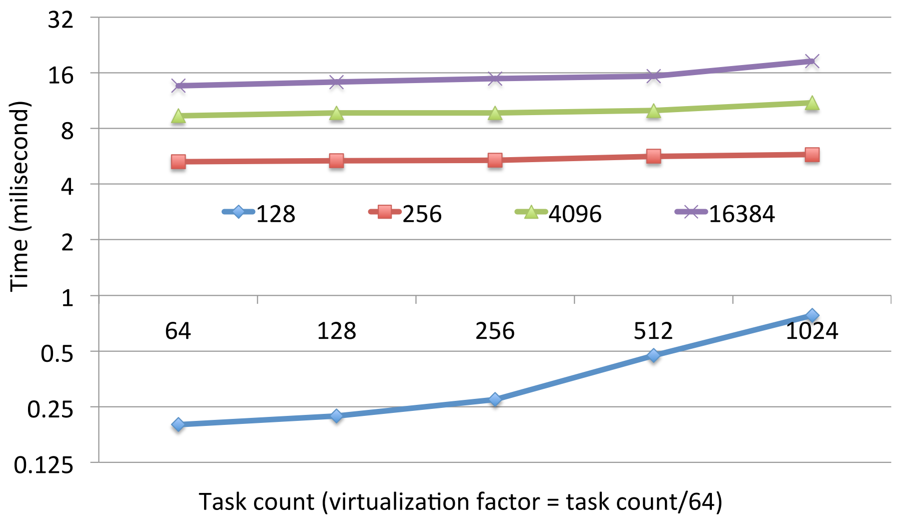


Figure 4.10. The effect of process virtualization on the scatter operation with various scatter sizes (in bytes) on 64 processor cores.

each process only needs to communicate with others in the same group.

Second, MPI allows the programmer to create a new communicator that modifies the order of MPI processes associated with the original communicator. In this scenario, the application performance may change, since reordering MPI processes may change the communication distance. For example, if process #0 frequently communicates with process #P-1, these 2 processes should physically reside in neighboring locations (e.g. compute nodes, processors). However, a default node allocation often creates a large distance between processes #0 and #P-1. The programmer can reorder the original ranks so that these 2 processes are physically close to each other. Another advantage of this technique is that the programmer does not have to modify the parallel algorithm.

MPI provides *MPI_Comm_split*, a common routine that can split an existing communicator into multiple disjoint groups, reorder MPI rank, or both. Its interface is as follows.

*int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *new-comm)*

MPI_Comm_split employs a color-key filtering mechanism. Shown in Fig. 4.11 is a 2-phase filter using *color* and *key*. In particular, color is a many-to-many mapping from the task ID set into the color set: color = c(id), where the color set is normally smaller than the task ID set. Key is a one-to-one mapping to sort tasks within a common color set: newIDcolor = k(id). In the example shown in Fig. 4.11, the *color* and *key* mappings are, respectively, c= id/2 and newId = id % 2.

Bamboo implements the *MPI_Comm_split* routine using MPI point-to-point primitives as follows. All MPI processes in the existing communicator exchange information of *color*, *key*, and the corresponding rank in *MPI_COMM_WORLD*. Eventually each process holds information of the other processes. We use a tree-based algorithm that requires each process to communicate with log(P) processes, where P is the number of processes within the existing communicator. Based on the information retrieved from others, each process filters out processes with the same color. Such processes will be sorted on *key* before being assigned a new rank in the new communicator. Once the new communicator has been created, a communicator name and a process rank within the communicator will be sufficient to locate the corresponding rank in *MPI_COMM_WORLD*. This allows the utility layer to employ the service provided by the *core message passing* layer.

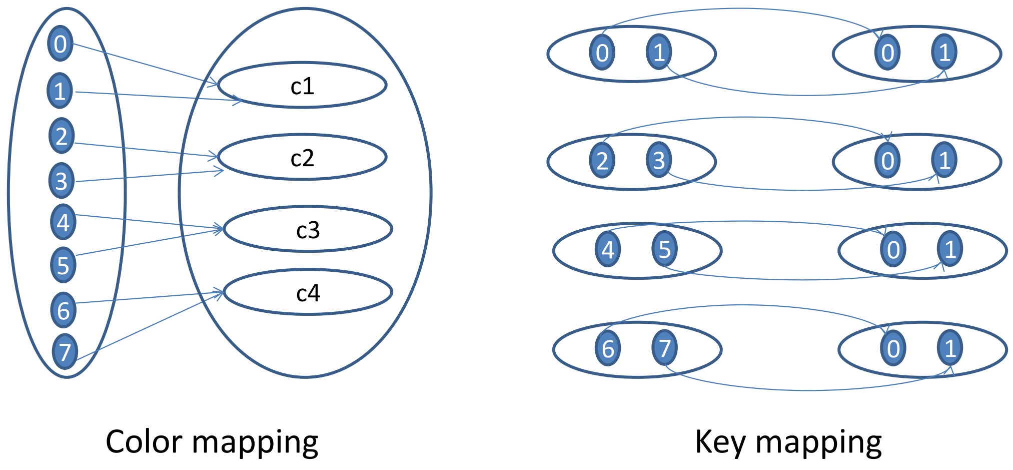


Figure 4.11. The MPI color-key filtering mechanism

4.5 Summary

1. The Bamboo source-to-source translator is layered as follows. The base layer of Bamboo handles MPI point-to-point primitives, whereas the utility layer on top of it supports MPI high-level routines. The implementation of the base layer is subject to services provided by a runtime system.

2. Tarragon is a runtime system providing services to operate a task dependency graph. Tarragon services include task scheduling and communication handling. In order to use these services, the programmer is supposed to implement the task behavior via overriding 3 methods defined by an abstract class called *Task*.

3. Bamboo employs the ROSE compiler framework to transform MPI code into a task dependency graph. A majority of the original MPI program is outlined into vinit() and vexecute() methods. The program control flow is reordered to conform to the Bamboo execution model. Send and receive calls are transformed into task inputs and outputs. Information associated with receive calls is also extracted to implement vinject() method.

4. MPI high-level routines such as collective operations and communicator splitting are broken down into point-to-point primitives. This approach allows the translation of these routines to be independent of the underlying runtime system.

4.6 Acknowledgements

This chapter, in part, is a reprint of the material as it appears in the article ”Bamboo - Translating MPI applications to a latency-tolerant, data-driven form” by Tan Nguyen, Pietro Cicotti, Eric Bylaska, Dan Quinlan and Scott Baden, which appears in

the Proceedings of the 2012 ACM/IEEE conference on Supercomputing (SC’12), Salt Lake City, UT, Nov. 10 -16, 2012.

Chapter 5

Dense linear algebra

5.1 Dense linear algebra in a nutshell

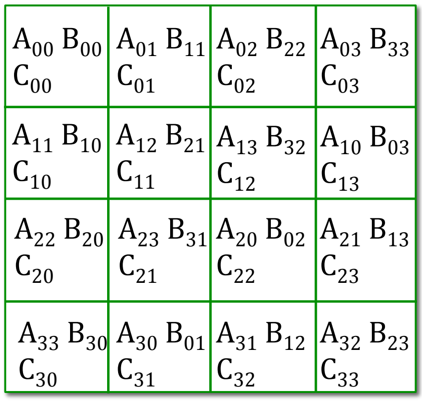
Dense linear algebra is a class of linear algebra computations on matrices where all elements are stored explicitly. Dense linear algebra arises in many scientific domains such as Quantum Chemistry [65, 66] and Computational Electromagnetics [67]. Typically this class of applications is highly amenable to acceleration. Thus, the overall performance will become much more sensitive to communication overheads as computing capability is expected to be substantially increasing in years to come. In this chapter, we evaluate Bamboo using matrix multiplication and matrix factorization, two operations commonly used as building blocks in many dense linear algebra problems.

5.2 Matrix-matrix multiplication

Matrix-matrix multiplication is a ubiquitous operation in linear algebra. In this dissertation, we consider *inner* and *outer product* formulations, two fundamental approaches to computing matrix multiplication. The former is employed in Cannon's algorithm [68, 69], whereas the latter is used in the SUMMA algorithm [2]. For both approaches, we rely on a serial matrix multiplication kernel called *dgemm*. The interface of *dgemm* is specified at level 3 of the Basic Linear Algebra Subprograms (BLAS) [70], a standard API for linear algebra.

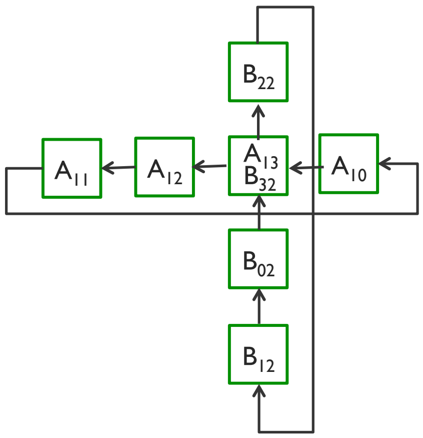
5.2.1 2D Cannon's algorithm

Cannon's algorithm employs a square process grid, such as the 4x4 grid shown in Fig. 5.1(a), to compute the matrix product of two matrices C=A*B [68, 69]. Each process owns a portion of the three matrices. The initial phase aligns sub-blocks of A and B so that their product corresponds to a sub-block of C. Fig. 5.1(a) presents the matrix alignment, which is also called *matrix skewing*. Under this alignment, A_{ij} is shifted leftward i process columns and B_{ij} is shifted upward j process rows. The algorithm then systematically rotates sub-blocks of A and B along process rows and columns in a sequence of $\sqrt{P}-1$ steps, where P is the number of processes. For example, Fig. 5.1(b) presents the matrix rotation performed by process P_{12} to produce the C_{12} sub-block. Within each rotation step, as soon as a process receives the next sub-block of A and B, it computes a partial matrix product to update the C partition that it owns ($C_{ij} += A_{ik} * B_{kj}$).



A_{00} B_{00} C_{00}
A_{01} B_{11} C_{01}
A_{02} B_{22} C_{02}
A_{03} B_{33} C_{03}
A_{11} B_{10} C_{10}
A_{12} B_{21} C_{11}
A_{13} B_{32} C_{12}
A_{10} B_{03} C_{13}
A_{22} B_{20} C_{20}
A_{23} B_{31} C_{21}
A_{20} B_{02} C_{22}
A_{21} B_{13} C_{23}
A_{33} B_{30} C_{30}
A_{30} B_{01} C_{31}
A_{31} B_{12} C_{32}
A_{32} B_{23} C_{33}

(a) Process P_{ij} owns C_{ij}. A and B are skewed



(b) Process P_{12} rotates and compute to update C_{12}

Figure 5.1. 2D Cannon's algorithm employs a square process grid to perform the matrix product C=A*B. A and B submatrices are aligned so that their product corresponds to C. The algorithm then systematically rotates A and B along process rows and columns.

Fig. 5.2 shows the basic MPI code of the 2D Cannon's algorithm. We call this variant *MPI-basic*. In this variant, MPI processes explicitly shift A and B partitions to neighbors using point-to-point messages. We annotated this code with Bamboo pragmas as follows. We marked the body of the *for* loop over the matrix rotation as an *olap-region*. Within this region, we placed *Irecv* calls in a *receive* block and *Send* calls in a *send* block. The reason for such annotation scheme is that each process sends and receives different data. The *Bamboo* code variant was generated by the Bamboo translator from the annotated MPI code shown in Fig. 5.2. We also implemented *MPI-olap*, a hand coded variant that overlaps communication with computation using a pipeline strategy. Fig. 5.3 shows the *MPI-olap* code, where we advanced MPI calls to enable overlap between the *dgemm* computation in one step and the matrix rotation of the next. Finally, *MPI-nocomm* code variant was obtained by suppressing all communication in the *MPI-basic* code.

```
dgemm(A, B, C);
#pragma bamboo olap
for(int step =1; step < sqrt(nprocs); step++){
  #pragma bamboo receive
  {  MPI_Irecv(rA from right);   MPI_Irecv(rB from down); }
  #pragma bamboo send
  {  MPI_Send(A to left);   MPI_Send(B to  up); }
  #pragma bamboo compute
  {MPI_Waitall();swap(A, rA); swap(B,rB); dgemm(A, B, C);}
}
```

Figure 5.2. Annotated code for submatrix rotation in Cannon's algorithm (*MPI-basic*). This annotated code will be transformed into *Bamboo*. *MPI-nocomm* is *MPI-basic* with MPI_Irecv, MPI_Send, and MPI_Waitall excluded.

```
for(int step =0; step < sqrt(nprocs); step++){
  if(step< sqrt(nprocs)-1){
    MPI_Irecv(rA from right);   MPI_Irecv(rB from down);
    MPI_Isend(A to left);   MPI_Isend(B to  up);
  }
  dgemm(A, B, C);
  if(step< sqrt(nprocs)-1){MPI_Waitall();swap(A, rA); swap(B,rB);}
}
```

Figure 5.3. We obtain *MPI-olap* by advancing message passing calls and overlap them with *dgemm*

We first conducted a weak scaling study on Hopper, a Cray XE6 cluster at

NERSC. Each compute node of Hopper consists of two AMD Magny-Cours 12-core processors, each including two 6-core sockets (see Appendix A for more specifications). Since Cannon's algorithm requires a square process grid, we ran with a square number of processes (i.e. 16 per compute node, 4 per socket, and 1 per processor core). We used *aprun* to launch jobs on Hopper. All MPI variants (*MPI-basic*, *MPI-olap*, and *MPI-nocomm*) were run with the following command line arguments, where *P* is the total number of cores: *-n P -N 16 -S 4*. To run the *Bamboo* variant, we employed P/4 MPI processes (one per socket), each containing 4 *worker threads*. The command line arguments to run the *Bamboo* variant is as follows: *-n P/4 -N 4 -S 1 -w 4*. We used square matrices of size up to 131,072x131,072.

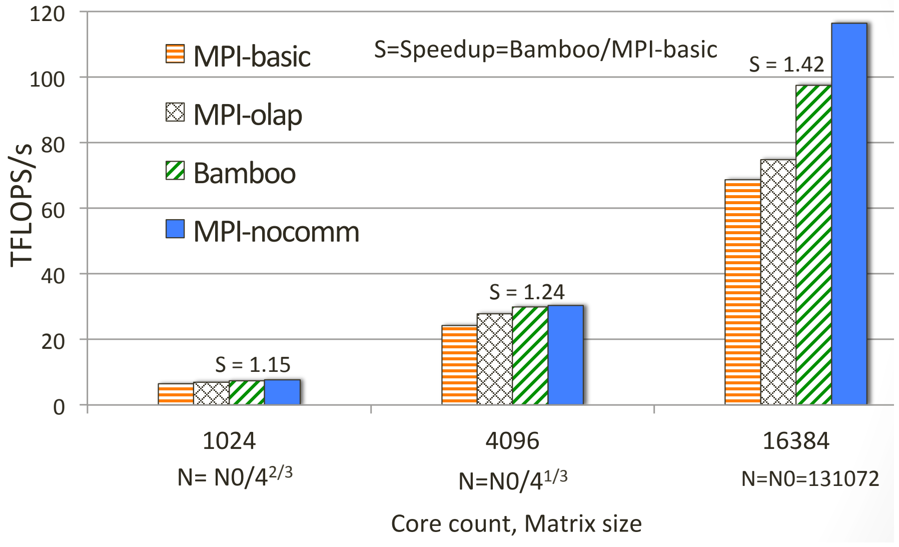


Figure 5.4. A weak scaling study on the 2D Cannon's algorithm on up to 16,384 processor cores on Hopper

Fig. 5.4 shows the performance of different code variants on up to 16K processors. From 1024 to 4096 cores, it can be seen that the communication cost increases steadily as the core count increases. The reason is as follows. The number of communication steps grows as $\sqrt{P}$. Since the wallclock time spent in *dgemm* remains constant, and the size of the local sub-matrices A and B grow as $P^{2/3}$, the communication to computation ratio grows as $P^{1/6}$. In fact, the observed growth in communication is a bit higher as we have ignored the increase in message starts, which also grow as $\sqrt{P}$. On 16,384 processor cores, however, the communication time increases significantly. This is a result of reducing pairwise communication bandwidth when the number of node increases. Under these conditions of growing communication costs, *Bamboo* improves the performance of the MPI-basic variant from 1.15 to 1.42 times, bringing performance closer to the upper bound (MPI-nocomm).

On Hopper, we found that the 2D Cannon's algorithm was not able to scale beyond 16,384 processor cores. In addition, the results on 16,384 cores on Hopper were quite sensitive to node allocation. As a result, we conducted another weak scaling study on up to 65,536 processor cores on Edison, a Cray XC30 cluster at NERSC. Each compute node of Edison consists of two 12-core Intel Ivy Bridge processors. Similar to the previous study on Hopper, we ran with a square number of processes on Edison (i.e. 16 per compute node, 8 per socket, and 1 per processor core). The MPI variants (*MPI-basic*, *MPI-olap*, and *MPI-nocomm*) were run with the following command line arguments, where *P* is the total number of cores: *-n P -N 16 -S 8.* To run the *Bamboo* variant, we employed P/8 MPI processes (one per socket), each containing 8 *worker threads*. The command line arguments to run the *Bamboo* variant is as follows: *-n P/8 -N 2 -S 1 -w 8*. We used square matrices of size up to 196,608x196,608.

Fig. 5.5 shows the performance of Bamboo and MPI variants on up to 65,536 processor cores. The results on Edison are much more stable than those on Hopper, though we conducted experiments at larger scales on Edison. However, comparing the performance of different code variants on Edison, we observed a similar trend that we had seen on Hopper. Specifically, *Bamboo* hides almost all of the communication overheads on 4096 and 16,384 processor cores, bringing performance closer to the upper bound (MPI-nocomm). *Bamboo* improves the performance of the *MPI-basic* variant 8% and

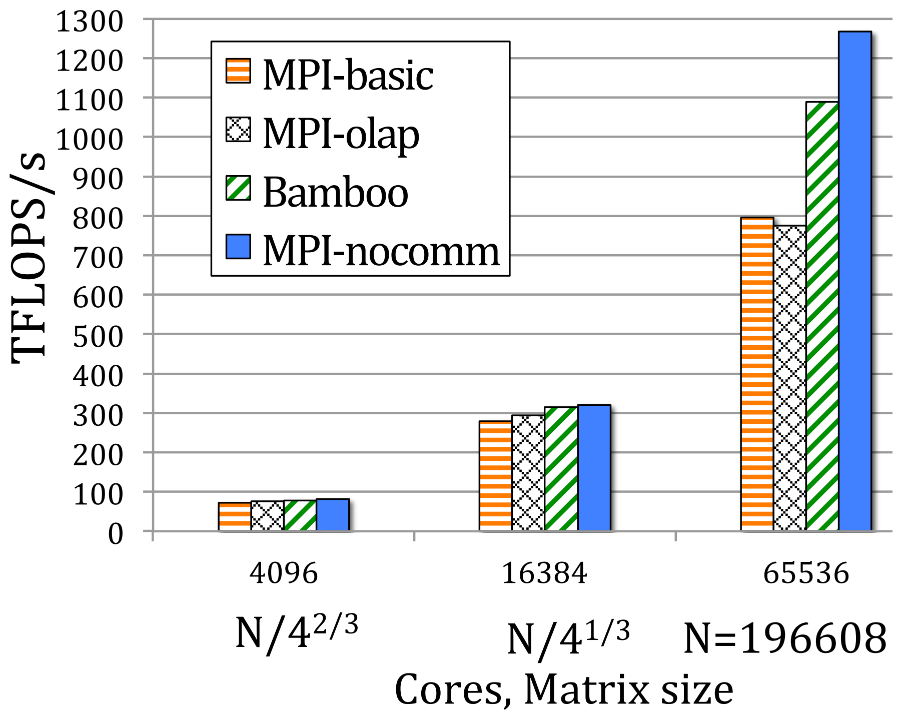


Figure 5.5. A weak scaling study on the 2D Cannon's algorithm on up to 65,536 processor cores on Edison

12.5% in time on 4096 to 16,384 cores respectively. On 65,356 processor cores, the communication time increases significantly to 37%. *Bamboo* increases the performance of the MPI-basic variant 1.37 times, corresponding to 27% improvement in time. The *MPI-olap* variant works well on up to 16,384 processors, but at 65,536 cores it cannot compete with *Bamboo*. We believe that this is due to the ability of Bamboo to use virtualization to better pipeline the communication.

5.2.2 2.5D Cannon's algorithm

We have just demonstrated that Bamboo is able to mask communication arising in multiplying large matrices. We next look at the *2.5D* Cannon (AKA communication avoiding) matrix multiplication algorithm [71], which targets small matrices. Small matrix products arise, for example, in electronic structure calculations (e.g. *ab-initio molecular dynamics* using planewave bases [72, 73]), a planned target of Bamboo.

The 2.5D algorithm is interesting for two reasons. First, small matrix products incur high communication costs relative to computation, especially at large scales, which stress Bamboo's ability to mask communication delays. Second, the 2.5D algorithm introduces two new communication patterns: broadcast and reduction. Supporting these new patterns broadens the scope of Bamboo.

At a high level, the 2.5D algorithm generalizes the traditional 2D Cannon algorithm by employing an additional process dimension to replicate the 2D process grid as shown in Fig. 5.6. The degree of replication is controlled by a replication factor called c. When c=1, we regress to 2D Cannon. When $c = c_{max} = nprocs^{1/3}$, we elide the shifting communication pattern and employ only broadcast and reduction. This algorithm is referred to as the 3D algorithm. The sweet spot for c falls somewhere between 1 and c_{max}, hence the name 2.5D algorithm.

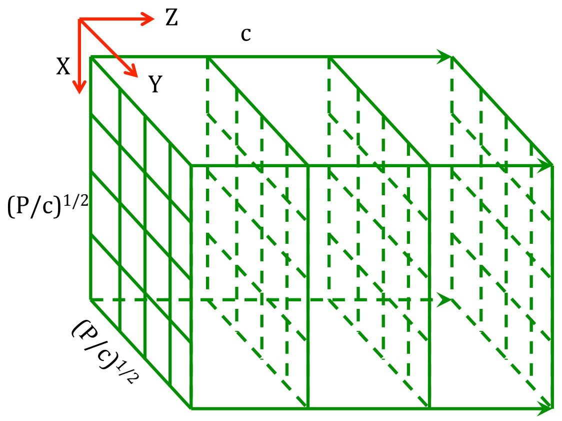


Figure 5.6. The 2.5D Cannon algorithm employs an additional process dimension to replicate the 2D Cannon process grid

As in the 2D algorithm, the 2.5D algorithm shifts data in the X and Y directions. In addition, the 2.5D algorithm performs a broadcast and a reduction along the Z dimension. Since broadcast and reduction are closely related, we show only the annotated code of the broadcast routine, in Fig. 5.7. Whereas the 2D algorithm uses a 2D process geometry, the 2.5D algorithm uses a 3D process geometry. Broadcast is based on a *min heap* structure

[74] constructed from the processes along the Z dimension. A min heap is a complete binary tree in which the parent's key (process ID in our case) is strictly smaller than those of its children. The broadcast algorithm has 2 communication blocks: one *receive* block and one *send* block. The *receive* block contains 1 *Recv* followed by 2 *Sends*. Since Bamboo will not reorder sending and receiving activities within a communication block it knows that the two *Sends* are dependent upon the completion of the *Recv*. However, following previous discussions about the independence of *send* and *receive* blocks, we infer from inspection that our annotations specify that all three point-to-point calls in the receive block are independent of all the point-to-point calls in the send block.

```
#pragma bamboo olap
{
  #pragma bamboo send
  {
    if(root & hasLeftChild) MPI_Send(A/B, leftChild);
    if(root & hasRightChild) MPI_Send(A/B, rightChild);
  }
  #pragma bamboo receive
  {
    if(!root & hasparent)  MPI_Recv(A/B, parent);
    if(!root & hasLeftChild) MPI_Send(A/B, leftChild);
    if(!root & hasRightChild) MPI_Send(A/B, rightChild);
  }
}
```

Figure 5.7. Annotated code for the broadcast routine employed in the 2.5D Cannon's algorithm.

Through experimentation, we observed that, with the small matrices targeted by the 2.5D algorithm, the hybrid execution model MPI+OMP yields higher performance than a pure MPI implementation, which spawns only one MPI process per core. Therefore, we used the following 3 variants: *MPI+OMP*, *MPI+OMP-olap*, and *Bamboo+OMP*. All variants perform communication at the node level, using the OpenMP interface of the ACML math library to multiply the submatrices (dgemm). *MPI+OMP* is the basic MPI implementation without any overlap. *MPI+OMP-olap* is the optimized variant of *MPI+OMP* that uses the pipeline strategy discussed previously for the 2D algorithm.

Bamboo+OMP is the result of passing the annotated *MPI+OMP* variant through Bamboo. As with the previous two applications, we also present results with communication shut off in the basic variant, i.e. *MPI+OMP-nocomm*, which uses the same code as *MPI+OMP*.

In the 2.5D algorithm the number of processes $P = 2^c q^2$ for integers c and q. Thus, the number of cores is a power of 2, and we used 4 cores per NUMA node. All variants spawned MPI processes at the NUMA-node level to take the advantage of node-level parallelism using OpenMP. We ran all variants with the following *aprun* command line arguments: *-n p4 -N 4 -S 1 -d 4 -ss*, where *p4 = P/4*. We set the environment variable OMP_NUM_THREADS=4 in all runs.

We conducted a weak scaling study on 4K, 8K, 16K and 32K processors on Hopper. We chose problem sizes that enabled us to demonstrate the algorithmic benefit of *data replication*.

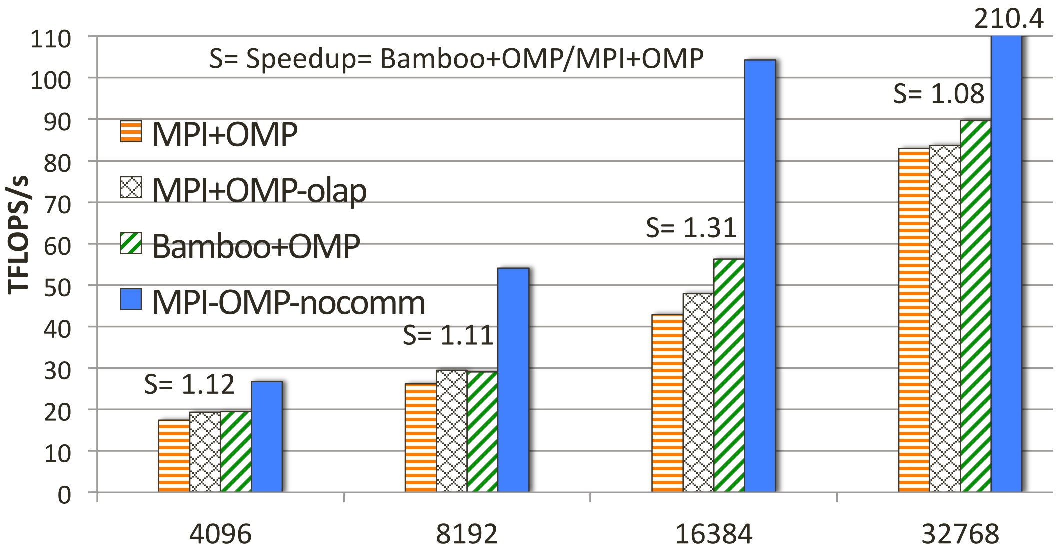


Figure 5.8. A weak scaling study on the 2.5D Cannon algorithm. We ran codes on up to 32768 processor cores on Hopper. We used small matrices (N=20668 on 4096 cores).

Fig. 5.8 shows the results with the different variants. We measured the communication cost, which ranged from 35% to 61% (wallclock time). Both *Bamboo+OMP*

Table 5.1. The effects of replication and virtualization. The *MPI+OMP* and *MPI+OMP-olap* code variants have limited options for c. The boldface values within the curly braces yield the highest performance.

#Cores	4096	8192	16384	32768
MPI+OMP	c= $\{\mathbf{1}, 4\}$	c= $\{\mathbf{2}, 8\}$	c= $\{1, \mathbf{4}, 16\}$	c=$\{\mathbf{2}, 8\}$
MPI+OMP-olap	c= $\{\mathbf{1}, 4\}$	c= $\{\mathbf{2}, 8\}$	c= $\{1, \mathbf{4}, 16\}$	c=$\{\mathbf{2}, 8\}$
Bamboo+OMP	c=2, VF =8	c=2, VF=4	c=2, VF=2	c=4, VF=2

and *MPI+OMP-olap* deliver the same speedup over the *MPI+OMP* variant on up to 8K processors. With 16K processors and more, *Bamboo+OMP* overtakes *MPI+OMP-olap*. Although *Bamboo+OMP* is still faster than the other variants on 32K cores, the speedup provided by *Bamboo+OMP* has dropped. We believe this behavior is the result of an interaction between the allowable replication factor c, and the degree of virtualization.

To understand the interaction, we first look at Table 5.1, which shows the values of c that maximize performance for the different variants. If we look into the performance of *MPI+OMP* variant on 8K, 16K and 32K cores, we notice that the efficiency suddenly drops on 16K cores but then increases again on 32K cores. This variation is likely the effect of replication. Note that the 2.5D algorithm requires that the first two dimensions of the processor geometry must be equal. For the two MPI variants (Table 5.1), the available values for the replication factor c are limited while *Bamboo+OMP* has more options due to the flexibility offered by virtualization. For example, on 8192 cores *MPI+OMP* and *MPI+OMP-olap* can set $c = 2$ or $c = 8$, i.e. other values are illegal. On 16K cores, c can be 1, 4 or 16 while on 32K cores c can take on values of 2 or 8. For *Bamboo+OMP*, performance depends not only on our choice of c but also on the degree of virtualization. Thus, we choose a combination of replication and virtualization that is optimal and cannot choose these parameters independently.

5.2.3 SUMMA algorithm

Scalable Universal Matrix Multiplication Algorithm (SUMMA) is another well-known parallel algorithm for matrix multiplication [2]. Unlike Cannon's algorithm and its variants, SUMMA does not require the process grid to be in any special shape. Thus, let PxQ be the grid of processes performing the matrix multiplication. The data decomposition of SUMMA is different from that of Cannon. In particular, matrices are first decomposed into submatrices in a block fashion. Each submatrix is then further decomposed into finer blocks called *panels*.

```
$C_{ij}$ = 0
for(k=0; k < K; k+=s){
   broadcast $A_{ik}$ to Q−1 processes in the same process row
   broadcast $B_{kj}$ to P−1 processes in the same process column
   $C_{ij} += A_{ik} * B_{kj}$
}
```

Figure 5.9. A high level description of SUMMA algorithm on a PxQ process grid

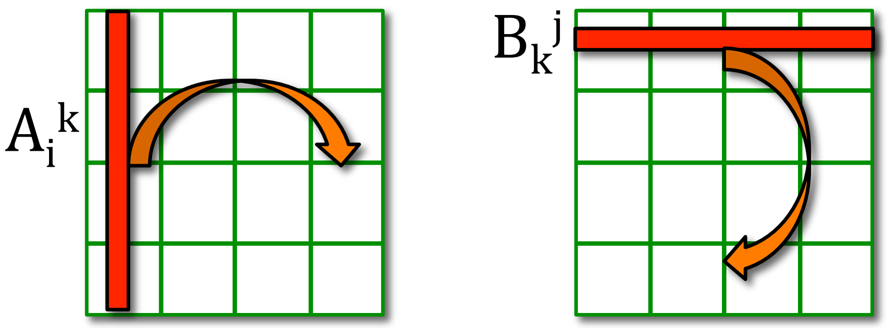


Figure 5.10. SUMMA is a broadcast-based algorithm. SUMMA broadcasts A along processes within the same row and B along processes within the same column.

Fig. 5.9 shows the high level description of SUMMA algorithm on this process grid. SUMMA is a broadcast-based algorithm, which can be briefly described as follows. The matrix multiplication $C_{MN} = A_{MK} \times B_{KN}$ progresses through K/s steps, where s is the *panel* size. In step *k*, processes owning column *panel k* broadcast their partition of this column (i.e. A_{ik}, i=0:P-1) to other processes in the same process row as shown in the left picture of Fig. 5.10. Likewise, processes owning row *panel k* broadcast their partition of this row (i.e. B_{kj}, j=0:Q-1) to other processes in the same process column as shown in the right picture of Fig. 5.10. Once a process P_{ij} receives column *panel* k and row *panel* k, it performs the submatrix multiplication to update C: $C_{ij} += A_{ik} * B_{kj}$.

Code variants

The basic MPI implementation used in this dissertation was developed based on the code provided by SUMMA's authors [2], and is shown in Fig. 5.11. This code employs ring broadcast operations as follows. A process broadcasting a *panel* does not have to send this *panel* to many other processes. Instead, it sends the A *panel* to the right neighbor and the B *panel* to the lower neighbor only. Receivers will help propagate messages until all processes have received what they need. During the broadcast of a *panel*, a process can perform the local submatrix update as long as all data required by such computation are available.

Figure 5.11 shows how we annotated the MPI code with Bamboo pragmas. Specifically, we marked the body of the *for* loop over the matrix multiplication as an *olap-region*. Within this region, we placed the *MPI_Send* calls performed by the roots of A and B *panels* in a *send* block. Non-root processes received A and B *panels* and forward them. Thus, the *MPI_Recv* and the following *MPI_Send* calls were co-located in the same *receive* block.

To realize overlap with Bamboo, it is generally required to employ more tasks than processor cores so that tasks can keep processor cores busy all the time. There are many possibilities for ordering tasks. Some orderings result in good performance while others result in poor performance. In SUMMA algorithm, a task after computing a *panel* can advance its computation to the next *panel* that it owns. At the same time, the neighboring task may have already received the message from this task. Thus, it is

```
$C_{ij}$ = 0
#pragma bamboo olap
for(k=0; k < K; k+= s){ //s is panelWidth
 #pragma bamboo send
 {
  if(I_own_A$^k$) MPI_Send $A_i^k$ to right
  if(I_own_B$^k$) MPI_Send $B_k^j$ to down
 }
 #pragma bamboo receive
 {
  if(I_do_not_own_A$^k$)                    MPI_Recv $A_i^k$ from left
  if(I_do_not_own_B$^k$)                    MPI_Recv $B_k^j$ from up
  if(my_right_neighbor_does_not_own_A$^k$) MPI_Send $A_i^k$ to right
  if(my_down_neighbor_does_not_own_B$^k$)  MPI_Send $B_k^j$ to down
 }
 $C_{ij} += A_i^k * B_k^j$
 updatePanelWidth(&s)
}
```

Figure 5.11. SUMMA algorithm employing ring broadcast operations [2]

also executable. In this situation, there is an advantage to schedule the neighboring task first, since doing so allows a task to forward received *panels* in a timely manner, thereby enabling other tasks. However, the task scheduler is oblivious to this information. Thus, we prioritized tasks as follows. In the horizontal dimension, the right task has higher priority than the left task. Likewise, in the vertical dimension, the lower task has higher priority than the upper task.

We ran the MPI and Bamboo code variants on the Edison cluster. Fig. 5.12 presents weak scaling results on up to 16,384 processor cores. It can be seen that, without prioritization, Bamboo slightly improves the performance of the MPI code variant. For this Bamboo variant, we employed only one task per process. Thus, the performance improvement can be attributed to the message buffering mechanism employed by Bamboo. In particular, in the original code the receiver handles one message at a time, and the next message has to wait until the local matrix multiplication completes. In Bamboo, the sender can send messages to a receiver without coordinating with the receiver. These messages will be buffered by the communication handler of the runtime system. Thus, when the receiver finishes its computation, it can quickly retrieves the required message

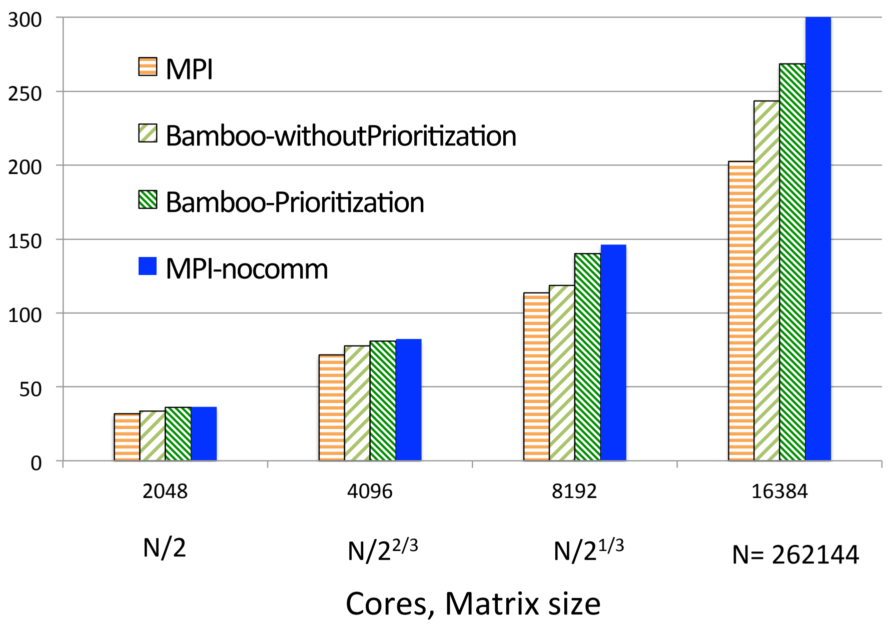


Figure 5.12. Weak scaling of SUMMA on up to 16,384 processor cores on Edison.

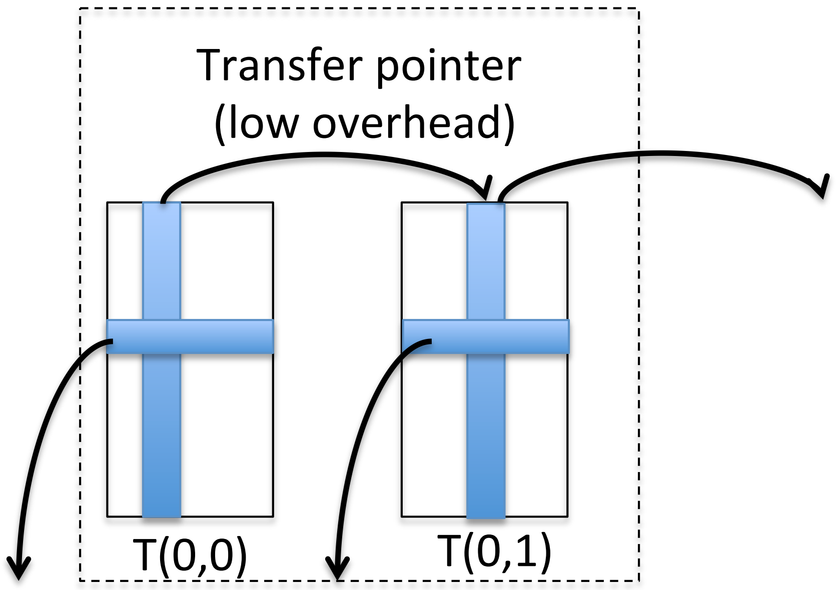


Figure 5.13. Virtualizing an MPI process into 2 tasks

from the message buffer to advance its execution.

When we use multiple prioritized tasks per MPI process (2, 4, and 8 tasks/process), the performance increases significantly. Fig. 5.13 depicts a process that contains 2 tasks. With the prioritization scheme presented ealier, a right task can have a better chance to forward messages from a left task. Thus, it allows messages to be broadcast more quickly.

5.3 Matrix factorization

5.3.1 LU factorization

LU factorization is a technique widely used to solve multiple systems of linear equations with the same left hand size. Specifically, LU factorization decomposes an input matrix A into a lower triangular matrix L and an upper triangular matrix U such that A = LU. Given L and U components of a matrix A, we can solve a system of linear equation Ax=b (i.e. LUx=b) by 2 steps: i) solve the lower triangular system Ly = b for y by forward substitution and ii) solve the upper triangular system Ux=y for x by back substitution. This dissertation uses a parallel blocked LU factorization algorithm with partial pivoting [56, 75].

We begin with the unblocked serial algorithm, to facilitate discussion of the blocked parallel algorithm. Fig. 5.14 presents the serial unblocked LU factorization code. The serial algorithm consists of n-1 stages, each corresponding to a column of the input matrix (line 2). A stage begins by identifying the element of the maximum magnitude in the portion of the column below the diagonal, the called the *pivot*. The row containing the *pivot* is then swapped with the current row *i*. Row *i* of matrix *L* and column *i* of matrix *U* are set on lines 5 and 6, respectively. Finally, we update the trailing submatrix below and to the right of the diagonal, expressed by a *rank-1 update* (line 7).

```
function [L,U] = LU(A)
  for i = 1:n−1
    v = max(abs(A(i:n,i))) %pivot
    A(i, i:n) ⟷ A(v, i:n) %swap 2 rows
    L(i:n, i) = A(i:n, i)/A(i, i)
    U(i, i:n) = A(i, i:n)
    A(i+1:n,i+1:n)=A(i+1:n,i+1:n) − L(i+1:n,i)*U(i,i+1:n)
  end
end
```

Figure 5.14. Serial algorithm for LU factorization

To improve locality, we use the blocked algorithm shown in Fig. 5.15(a). Except for pivot selection and row interchange, the blocked algorithm updates the L, U, and A at the block granularity. Since the trailing submatrix of A shrinks as factorization proceeds to the right, a parallel implementation maps the blocks cyclically to processes (blocked cyclic decomposition) to balance the workload. As shown in Fig. 5.15(b), the matrix A is decomposed into 8x8 blocks (AKA panels), which are then distributed cyclically onto a 2x4 process geometry. We next describe the distributed-memory algorithm for parallel LU factorization, based on message passing.

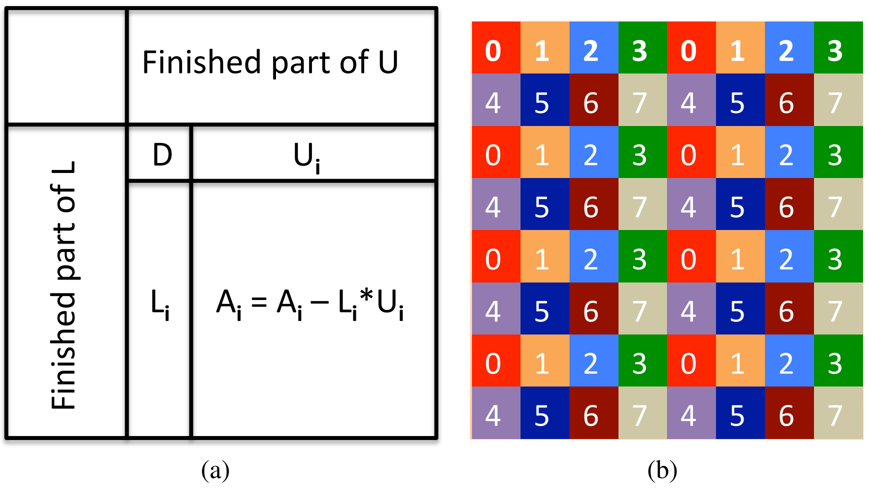


Figure 5.15. A blocked, right looking algorithm (a) and a blocked cyclic decomposition of LU factorization on 8 processes

5.3.2 High Performance Linpack

We used the High Performance Linpack benchmark (or HPL, or Linpack for short) [55–57]. HPL is a well-known benchmark written in C that solves a dense system of linear equations using LU factorization, and is often used to measure the performance of newly constructed systems. The application employs a blocked cyclic data decomposition scheme as previously discussed (see Fig. 5.15(b)).

The HPL benchmark comprises 2 code variants. *Pdgesv0* does not make any attempt to overlap communication with computation, whereas *pdgesvK2* applies an overlapping technique called *lookahead.* We applied Bamboo annotations to *pdgesv0*. Details of the 3 code variants are as follows. The pseudo code for *pdgesv0* appears in Fig. 5.16. The code consists of 3 key operations: panel factorization (*pFact* on lines 11-17), panel broadcast (*pBcast* on lines 18-26), and the trailing submatrix update (*pUpdate* on lines 27-30). *pFact* finds the *pivots* in column panel *c*. This step is costly since we have to factorize a skinny matrix over a subset of the processes that own the panel (the regions D and L_i in Fig. 5.15(a), including a sequence of row swap-broadcasts, one for each *pivot* within a single columns of the panel. HPL provides various panel factorization implementations, classified into recursive and non-recursive variants. We evaluated both variants and observed no difference in performance. Thus, we used the non-recursive variants. Once the panel has been factorized it must be broadcast to column processes within the same row *(pBcast).* HPL provides many algorithms to perform the panel broadcast. We used the *HPL_bcast_1ring* broadcast algorithm for panel broadcast. This is an efficient implementation that uses a ring broadcast algorithm, shifting data to the right along column processes. The *pUpdate* operation swap-broadcasts U among row processes and then performs a *rank-1 update*. It accounts for the lion's share of LU's computational work, performing $O(N^3)$ multiply-adds.

The *pdgesvK2* variant applies *lookahead* [55, 76], a technique for overlapping communication with computation that fills idle gaps in the execution of LU. *Lookahead* utilizes the dependence structure of the blocked algorithm to orchestrate computation and data motion. It uses split-phase coding [13], and may compute multiple iterations in advance. Fig. 5.17 shows a simplified implementation of the *lookahead* algorithm where the *lookahead* depth is one, and a simplified strategy is used to synchronize communication. Once a process receives a panel, it performs a partial update so that it can factorize the next panel. While the panel factorization step is communicating, and during the panel broadcast of the next panel, the process can perform useful work in the remaining update.

```
function [L, U] = nolookahead_LU (A)
  for c =  1:columnPanels
    if isMyPanel(c) == true  % if I own the panel c
      L = pFact(c, columnComm)
    end
    L= pBast (L, c, rowComm)
    [A, U] = pUpdate (A, L, c)
   end
end

function L = pFact(c, comm)
  for j= 1:panelSize
    find_pivot(c, j, comm) %reduce local pivots in comm
    swap-bcast(c, j, comm)
    L[j] = update_L(j)
  end
end
function L= pBcast(P, c, comm)
  if isMyPanel(c) == true
    shift P to the right neighbor
  else
    receive P from the left neighbor
    shift P to the right neighbor
  end
  L = P
end
function [A, U] = pUpdate (A, L, c)
  U= update_U(c) %broadcast among rows
  update trailing submatrix // A = A - L * U
end
```

Figure 5.16. Message passing implementation of LU factorization without *lookahead*. The application divides into 3 separate operations: panel factorization, panel broadcast, and the trailing submatrix update.

```
function [L,U] = LU_lookahead(A)
  if isMyPanel(1) == true
    $L_1$ = pFact(1, columnComm)
    shift $L_1$ to the right neighbor
  end
  for c = 1:columnPanels − 1
    if isMyPanel(c) == false
      receive $L_c$ from the left neighbor
      shift $L_c$ to the right neighbor
    end
    partial_pUpdate(A, $L_c$)
    if isMyPanel(c+1) == true
      $P_{c+1}$ = factorize(c+1) //pivoting and updating L
      shift $P_{c+1}$ to the right neighbor
    end
    remaining_pUpdate (A, $L_c$)
  end
  if isMyPanel(columnPanels) == false
    receive and forward $L_{columnPanels}$
  end
  pUpdate(A, $L_{columnPanels}$)
end
```

Figure 5.17. A simplified parallel algorithm of LU factorization with *lookahead* (no message probing and depth=1)

We annotated the *pdgesv0* module and translated it with Bamboo. After the task graph code has been generated we compiled and ran it like any conventional ordinary C++ program. We also added scheduling policies via task prioritization so that communication could be overlapped with communication more efficiently. The common wisdom in scheduling a non-preemptive task graph is that tasks should hold the processor/core as long as they are still executable and only yield control when they need input from other tasks. This greedy strategy is intended to maintain the high hit rates of caches and TLB. However, scheduling LU factorization is an exception. Specifically, many tasks are waiting for data from the root task so that they can begin executing. Moreover, if for some reason the task that will become the next root is not scheduled soon, the next panel broadcast will be delayed. If this happens, performance could be significantly penalized since no overlap can be realized.

Bamboo's *olap-regions* generally reside within an outer iteration, and HPL is no exception. Bamboo handles overlap regions as follows. When control reaches the end of an overlap region, if the priority is negative, the task yields processor/core, even if inputs are ready for the next iteration. To this end, we used 3 different values 0, -1, and 1 to represent for the urgency of scheduling a task. Among runnable tasks, those with higher priorities will be inserted at the top of the priority scheduling queue. Tasks with priority of 0 or 1 will execute until they cannot continue, since they await data from other tasks that haven't yet completed. However, tasks with priority -1, must yield processor/core at the end of the olap region, even if they have the data needed to continue executing. Note that this scheme is not preemption. Neither the runtime system nor task can force another task to yield control. Depending on the availability of the input and the current priority, a task decides whether it should continue or yield processor/core to another task.

Fig. 5.18 shows how we prioritized LU tasks. By default all tasks have priority 0. At the panel bcast, the root sends out a message and reduces its priority. Finally, if a task becomes the next root, it will increase its priority after the bcast message has been transmitted to the next task in the row.

```
function taskBcastInit(Panel, root, comm)
  if _taskID == root
    put(Panel, right_neighbor, comm)
    _state = EXEC % just means that task is runnable
    _priority = -1  %root is no longer urgent
  else
    task.priority = -1
    if task.previous == root  % I will become the new root
      Remember to set my priority to 1 once the intra-node bcast is done
    end
    _state = WAIT %return the processor/core and wait for data
  end
end
```

Figure 5.18. Setting priorities in the task graph.

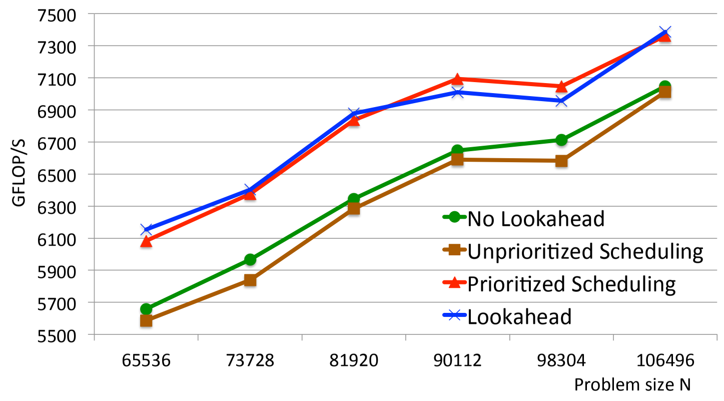
5.3.3 Performance evaluation

We performed experiments on Stampede [77], a system located at the Texas Advanced Computing Center (TACC). Stampede consists of 6400 compute nodes, each equipped with two 8-core Sandy Bridge processors and one Intel Xeon Phi XE10P coprocessor (MIC). We only used the Sandy Bridge portion since sophisticated optimizations are required to offload and execute code modules efficiently on MIC [78]. For more specifications of this platform, see Appendix A.

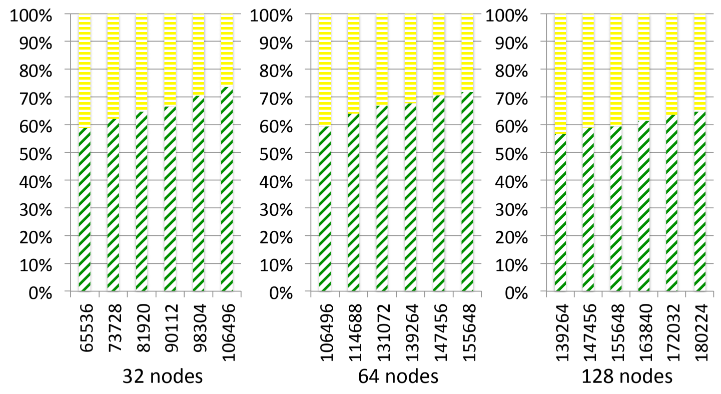
Code variants and problem sizes

We evaluated 4 code variants of the HPL benchmark. The first 2 variants are the original MPI code with and without the *lookahead* optimization (i.e. *pdgesvK2* and *pdgesv0* respectively). The third and fourth variants are the result of passing *pdgesv0* through the Bamboo translator. The fourth variant was obtained by making modest changes to the third variant, that provide hints to the scheduler to prioritize tasks that result in improved overlap. These changes come in the form of *performance meta-data* [79], which are annotations to the task dependency graph that are seen by the controller. The meta-data are abstract entities (i.e. integers) so the scheduler and application are unaware of one another. Thus, the programmer is free to interpret the meaning of meta-data, which provide convenient mechanism for exploring application specific scheduling.

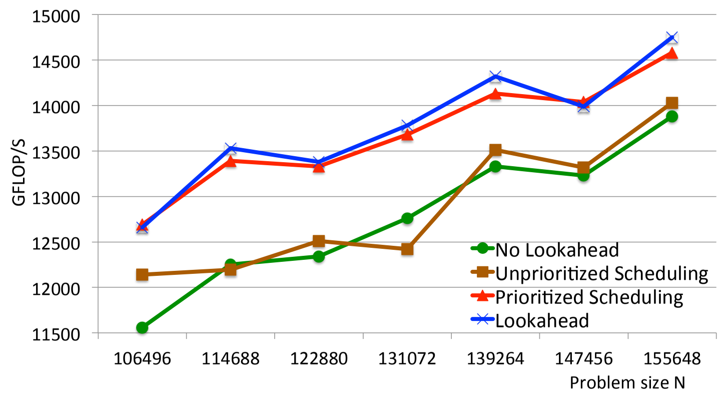
Fig. 5.19 presents results of running different code variants on up to 128 compute nodes on Stampede. For these experiments, we selected small enough problem sizes so that the communication overhead is significant and thus we can see the benefit of overlapping communication with computation. The insight of using small problem sizes is that computation grows faster than communication ($2/3 * n^3$ as opposed to $1/2 * n^2$); thus, the smaller the problem size, the larger overhead of moving data relative to computation. However, problem sizes also need to be large enough so that overheads introduced by the compiler and its runtime system can be amortized.



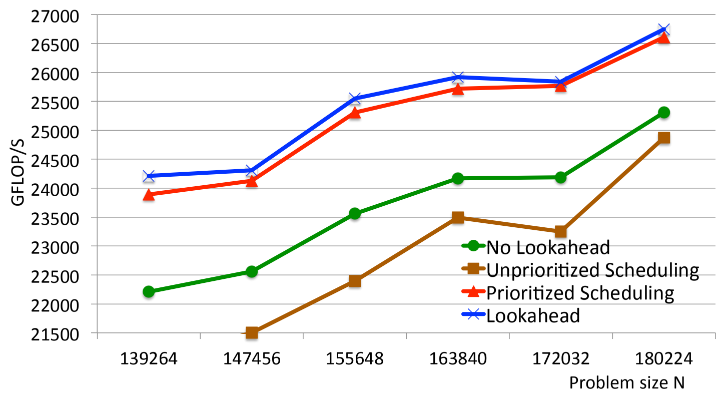
(a) 32 Stampede nodes



(b) Time distribution. In all experiments, time for dgemm (green, lower bars) varies from 57% to73%. Panel broadcast accounts for most of the rest of the time.



(c) 64 Stampede nodes



(d) 128 Stampede nodes

Figure 5.19. Results comparing our scheduling strategies for the transformed code without lookahead and with lookahead. Prioritization significantly improves performance, enabling our transformed code to meet the performance of lookahead for many problem sizes

Panel size and process geometry

As mentioned in Section 5.3.1, the panel size has to be small enough to balance the workload, but large enough to maintain the efficiency of local computations. We manually tuned this parameter and found that a panel size of 96 delivered optimal performance for the selected problem sizes we used on Stampede. The performance of HPL is also sensitive to the process geometry. Specifically, if the width of the process grid is too large (i.e. too many processes within a process row), the panel broadcast will be costly. However, if the height of the process grid is too large, the overheads of the panel factorization and U broadcast will be substantially increased. As a result, we always choose a nearly square process grid with a constraint that the size of each dimension is a power of 2. Note that in the task graph variant, we virtualized process rows only. Thus, the width of the task graph is always 4 times as large as the height. The reason is that our task scheduling algorithm is to overlap the panel broadcast, which lays on the critical path.

Analysis of results

It can be seen in Fig. 5.19 that the *lookahead* and prioritized Task Graph variants always outperform the *no-lookahead* one on every problem size and on any number of nodes. For a fixed number of nodes, the performance improvement is more significant with small problem sizes. On 32 nodes the benefit of overlap is 8% with the smallest value of N but only 4% with the largest N. Similarly the benefit ranges from 10% to 6% on 64 nodes and 8% to 5% on 128 nodes. Reducing the problem size further, however, may decrease the performance benefit since we would not have enough computation to overlap with communication. Fig. 5.19b shows the amount of computation (dgemm) that can be used to hide communication. Dgemm computation accounts for 58%-73% of the total execution time. These results also validate our earlier analysis, that the relative overhead of communication shrinks as the problem size grows.

The vital role of task prioritization is inevitable. Theoretically, if we use a random scheduling algorithm and we repeat the experiment with the unprioritized Task Graph variant for a large number of times, there is possibility that we experience the performance of the prioritized Task Graph variant. However, the required number of experiments could grow exponentially due to the enormous space of scheduling decisions. Specifically, assuming that there are always at least 2 tasks (among many tasks per MPI process) being ready to execute, the number of scheduling decision is $O(2^{k*N})$, where N is the number of panel columns of the input matrix and k is the number of communication events occurring with a particular N. We repeated experiments for a few times, but results without task prioritization were far below the performance of *lookahead*. Compared to the *no-lookahead* variant, the performance of the unprioritized task graph was at best comparable and in some cases it was even lower. Fig. 5.19 shows that the performance of the prioritized Task Graph variant meets and sometimes slightly exceeds the performance of *lookahead* on most problem sizes. This result is likely to hold with larger problem sizes and is independent of the number of nodes.

5.4 Summary

Dense linear algebra is an important class of scientific computation. This dissertation employs two common operations of dense linear algebra to validate Bamboo: matrix multiplication and matrix factorization. The results are as follows.

1. Bamboo significantly improves the performance of 2D Cannon's algorithm, a widely used matrix multiplication algorithm, on up to 16,384 processor cores on Hopper and 65,536 processor cores on Edison. For the communication avoiding (2.5D) variant of the Cannon's algorithm, Bamboo is able to improve performance

via overlapping communication with computation, even though the algorithm avoids communication. Both techniques play an important role in highly scalable computing, where we must take into account a variety of performance tradeoffs, some of them algorithmic. For the SUMMA algorithm, Bamboo enables additional overlap by providing the message buffering mechanism and task prioritization.

2. Hiding communication overheads arising in LU factorization is challenging due to the cyclic decomposition and the broadcast-based algorithm. *Lookahead* is a well-known technique for masking communication in matrix factorization arising in linear algebra, but at a cost of added software complications. Bamboo allows the programmer to translate MPI source without *lookahead* into a task graph representation that can automatically overlap communication with computation. The programmer can also embed scheduling heuristics into the program execution via task priority. Experimental results demonstrated that scheduling the LU task graph with task prioritization significantly increases the performance.

5.5 Acknowledgements

This chapter, in part, is a reprint of the material as it appears in the article ”Bamboo - Translating MPI applications to a latency-tolerant, data-driven form” by Tan Nguyen, Pietro Cicotti, Eric Bylaska, Dan Quinlan and Scott Baden, which appears in the Proceedings of the 2012 ACM/IEEE conference on Supercomputing (SC’12), Salt Lake City, UT, Nov. 10 -16, 2012.

This chapter, in part, is currently being prepared for submission for publication with Scott Baden. The dissertation author is the primary investigator and author of this material.

Chapter 6

Structured grid

6.1 Overview

Structured grid (SG) methods employ regular patterns, called a grid, to discretize a physical domain into a finite set of elements. Grids in SG methods often contain quadrilateral elements in 2D and hexahedral elements in 3D. Using a constructed structured grid, one can approximate the solution of a partial differential equation (PDE) within the domain. Such a grid may comprise not only a single block of identical elements but also multiple blocks containing elements of different discretization spacing. When multiple blocks are present, the method is called *block structured grid*. Whether *structured grid* or *block structured grid* methods are used, the programmer has the flexibility to position the grids to fill the physical domain. In this chapter, we use Bamboo to speed up a solver for Poisson's equation using the iterative Jacobi method on a single-block grid [17–20] and a solver for Helmholtz's equation using a multigrid method [3].

6.2 3D Jacobi solver

We use 3D-Jacobi, an iterative solver for Poisson's equation in three dimensions $\nabla^2 u = f$, subject to Dirichlet boundary conditions. The solver employs Jacobi's method with a 7-point central difference scheme that updates each point of the grid with the

```
// V, U, and rhs are N x N x N grids
for step = 1 to num_steps{
 for k = 1 to N−2 //Z
  for j = 1 to N−2 //Y
   for i = 1 to N−2 //X: the leading dimension
     V[k,j,i]= alpha *(U[k−1,j,i]+U[k+1,j,i]+U[k,j−1,i]+U[k,j+1,i]+U[k,j,i−1]+U[k,j,
          i+1])−beta*rhs[k,j,i]
  swap(U,V)
}
```

Figure 6.1. Serial kernel of 3D Jacobi

average of the six nearest neighbor values in the Manhattan directions [17–20].

The main kernel of 7-point stencil comprises a 4-level nested loop as shown in Fig. 6.1. The outermost loop enumerates on a discretized time domain whereas the remaining three loops sweep the 7-point stencil operation over the 3D discretized spatial domains with coordinates X, Y, and Z. The data consists of 3D arrays, stored in row major order. We apply spatial blocking, dividing the discrete domain into many small tiles so that the working set of each one fits on cache. In particular, we performed a 2D spatial blocking for L2 cache along the Y and Z axes. We determined experimentally that a 4x8 block size was optimal. Compared to other operations in the stencil family, the 7-point stencil has a low flops/write ratio, which challenges the ability to overlap computation with communication.

6.2.1 Code variants

In order to make fair performance comparisons, we compared several variants of 3D Jacobi. All variants share the same numerical kernels. The first variant, *MPI-basic*, is the simplest. It does not overlap communication with computation and is the starting point for the remaining variants. This code was previously shown in Fig. 3.8. The second variant, *MPI-olap*, has been manually restructured to employ *split-phase* coding to overlap communication with computation. Specifically, it employs a hierarchical data decomposition, subdividing the mesh assigned to each core into 8 equal parts using a 3D 2x2x2 geometry. *MPI-olap* sets up a pipeline; within the outer iteration it sweeps one-half of the 8 sub-problems while communicating ghost cells for the others. The third variant, *MPI+OMP*, employs a hybrid execution model running 1 MPI process on a set of processor cores. Each process unfolds a team of OpenMP threads to perform the mesh sweep.This hybrid variant uses just a fraction, 1/T, of the MPI processes used in the pure MPI variant, where T is the number of OpenMP threads per process. Under these conditions communication occurs at the process level: the single master thread exchanges ghost cells between processes leaving all but one core idle during communication. The fourth variant, *MPI+OMP-olap*, combines the overlapping technique used in the second variant with the hybrid model used in the third. This variant takes advantage of both techniques, though the hierarchical control flow may reduce the effectiveness of overlap. The fifth and the sixth variants, *Bamboo-basic* and *Bamboo+OMP*, were obtained by passing the *MPI-basic* and *MPI-OMP* variants, respectively, through the Bamboo translator. These codes run in data-driven fashion under the control of the Tarragon runtime system. The Bamboo annotations used in 3D Jacobi are similar to the code previously shown in Fig. 3.8. We also report performance for *MPI-nocomm*. This is not a true variant and is the result of turning off all message passing activity in *MPI-basic*. We use *MPI-nocomm* to establish an upper bound on performance, which we may or may not be able to realize in practice. Since 3D Jacobi is a memory bandwidth-bound application, the performance of *MPI-nocomm* is far below the peak performance of the hardware.

6.2.2 Performance evaluation

We conducted a strong scaling study on Hopper, a Cray XE6 cluster at NERSC. Strong scaling stresses communication overhead, though we still have sufficient computation to overlap with data motion. We maintained the problem size at 3072^3 as we increase the number of processors. All jobs were launched using the *aprun* command. The pure MPI variants (*MPI-basic* and *MPI-olap*) ran with 1 process per core, while the others ran with 1 process per NUMA node, each spawning an identical number of OpenMP threads. Thus, the MPI variants were run with the following *aprun* command line arguments *-n P -N 24 -S 6,* where *P* is the total number of cores and we run with 24 MPI processes per Hopper node (*-N 24*) further organized into four groups of 6 processes per NUMA node (*-S 6*). The other variants ran with one MPI process per NUMA node using the following *aprun* command line arguments: *-n p6 -N 4 -S 1*, where *p6 = P/6*. For the hybrid variants using OpenMP (*MPI+OMP*, *MPI+OMP-olap*, and *Bamboo+OMP*), we specified *-d 6* to spawn 6 worker threads per NUMA node. For the Bamboo variants of the pure MPI codes, the translator manages thread spawning via Tarragon. It configured Tarragon to spawn 5 worker threads, each running on its own core, dedicating the remaining core to a service thread. Lastly, we specified the -ss option of *aprun*, which restricts each thread to use memory nearest to its NUMA node, improving performance.

Fig. 6.2a compares the results with different variants of 3D Jacobi. Notably, *Bamboo* uniformly improves performance of both variants (*MPI-basic* and *MPI+OMP*) at all levels of parallelism. For example, on 96K (98034) cores, *Bamboo-basic* realizes a $\times 1.27$ speedup, hiding 52% of the communication delay in *MPI-basic.* More generally, the speedups ranged from 1.07 to 1.27. With strong scaling, communication overhead increases with the number of cores (from 13% to 41% over the range of 12K to 96K cores), and this explains why the performance increase delivered by Bamboo grows with the number of cores. Since the kernel is blocked for cache in all variants, we believe that most of the benefits come from latency hiding. To gain insight into the performance benefits of Bamboo, we next analyze the remaining two MPI variants.

The hybrid *MPI+OMP* variant demonstrates the benefits of multithreading which is also enjoyed by the *Bamboo* variants. Though this hybrid variant provides only a

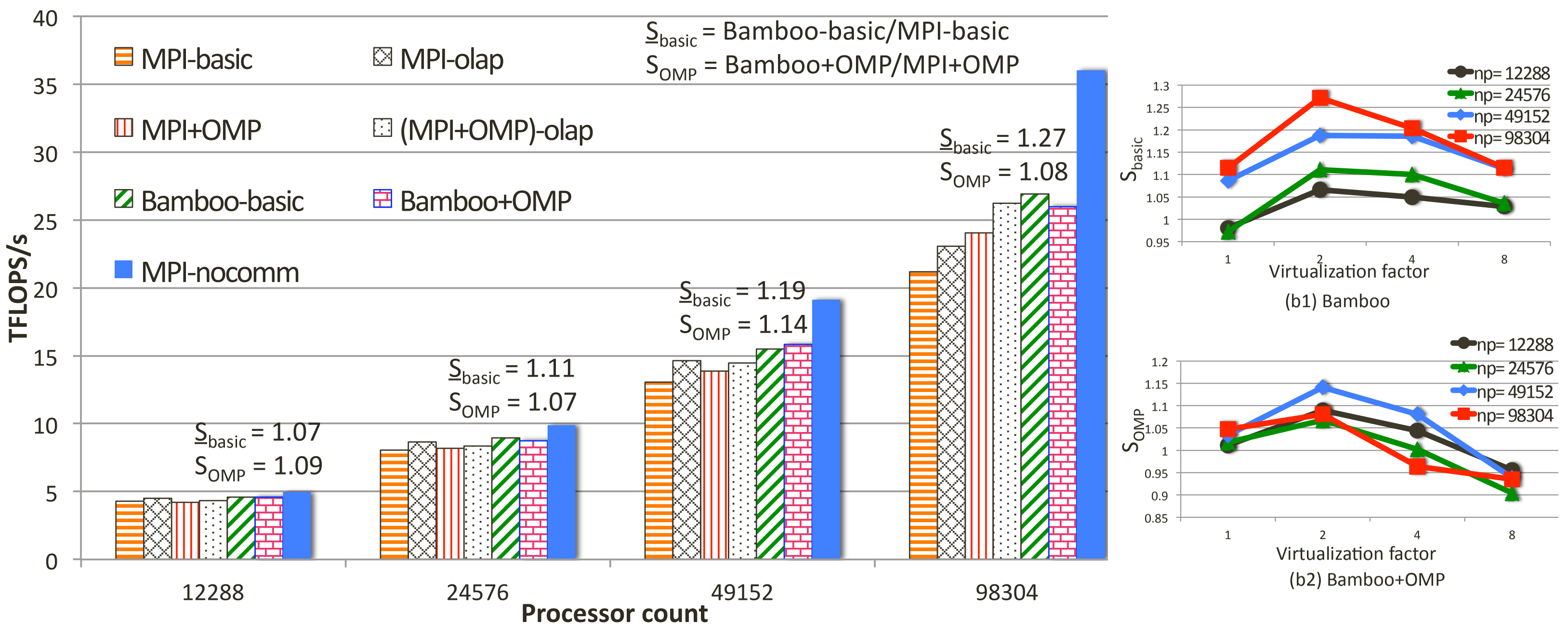


Figure 6.2. 3D Jacobi. Grid size: 3072x3072x3072 double precision.

modest improvement over *MPI-basic* on smaller numbers of cores, it provides a large boost at 96K cores. We believe this is due to reduced communication delays achieved by hybrid MPI-thread execution at scale, which is also exhibited by Bamboo. In our strong scaling study, messages are shrinking from 192KB to 24KB as the number of cores increases from 12228 to 98304. Since only 1 MPI process per NUMA node is communicating, *MPI+OMP* and *Bamboo+OMP* variants aggregate the shorter messages into a smaller number of longer messages, compared to 1 MPI process per core with the pure MPI implementations. Since the network interface serializes messages longer than the eager limit, it makes sense that aggregation should benefit performance. However, why this effect appears suddenly at only at 96K cores is as yet unclear, and is currently under investigation. An outstanding difficulty is that our 96K core jobs allocate 64% of the machine, and long queue delays hamper experimentation.

Bamboo and the hand optimized *MPI-olap* variants deliver similar performance at up to 24K cores, but on 48K and 96K cores Bamboo's advantage rises sharply. We attribute the sudden change to how Bamboo handles decomposition. *MPI-olap* uses a hardwired scheme of 8 tasks per core, splitting the mesh assigned a core along all 3 dimensions. Bamboo, on the contrary, has more flexibility than *MPI-olap* in selecting task geometry as it can virtualize along any of the dimensions. We experimented with many geometries and found that a virtualization factor of 2 tasks per core was optimal (Fig. 6.2.)

Bamboo-basic generally outperforms *Bamboo+OMP* since the runtime services run independently on one core while they have to share a processor with an OpenMP thread in *Bamboo+OMP*. This is revealed in Fig. 6.2, which also shows that the benefits of communication-computation overlap in *Bamboo+OMP* drop off more quickly than Bamboo-basic as we increase the virtualization factor. Both variants benefit from modest amounts of virtualization (2 tasks per core), which improves the pipelining of communication and computation, but higher levels of virtualization introduce increased scheduling costs, overwhelming any improvements due to overlap.

6.3 Multigrid solver

6.3.1 Multigrid solver

Multigrid [80–82] is a family of methods to accelerate the convergence rate of conventional iterative methods such as the Jacobi method used in Section 6.2. A multigrid solver consists of multiple cycles. In each cycle the solver keeps restricting the problem into coarser grids, then it conducts an iterative or exact solver at the bottom level before interpolating the solution back to the finer grids. The cycle can be in V or W shape, or can be truncated at a certain level where the bottom solver can perform efficiently. Figure 6.3 shows a truncated V-cycle, where the restriction process stops at the 4th grid level.

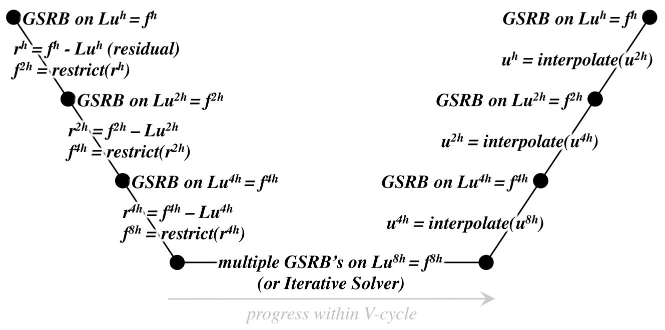

GSRB on $Lu^h = f^h$
$r^h = f^h - Lu^h$ (residual)
$f^{2h} = restrict(r^h)$
GSRB on $Lu^{2h} = f^{2h}$
$r^{2h} = f^{2h} - Lu^{2h}$
$f^{4h} = restrict(r^{2h})$
GSRB on $Lu^{4h} = f^{4h}$
$r^{4h} = f^{4h} - Lu^{4h}$
$f^{8h} = restrict(r^{4h})$
multiple GSRB's on $Lu^{8h} = f^{8h}$
(or Iterative Solver)
progress within V-cycle
GSRB on $Lu^{4h} = f^{4h}$
$u^{4h} = interpolate(u^{8h})$
GSRB on $Lu^{2h} = f^{2h}$
$u^{2h} = interpolate(u^{4h})$
GSRB on $Lu^h = f^h$
$u^h = interpolate(u^{2h})$

Figure 6.3. A truncated V-cycle for solving $Lu^h = f^h$, where h is the grid spacing (image source [3]).

6.3.2 Code variants

We translated a multigrid solver developed at Lawrence Berkeley National Laboratory to solve Helmholtz equation [3]. This is an MPI+OpenMP code consisting of 4000 lines, 1000 of which are MPI code that need to be translated. This solver employs *truncated V-cycles*. On the way down of each cycle, *smooths* are applied to reduce the error before *restrictions* are used to determine the right-hand side of the coarser grids. Each *smooth* is a Gauss-Seidel Red-Black relaxation (GSRB), which delivers faster convergence rate than other techniques of the same type such as Jacobi and Successive Over-Relaxation [83]. V-cycle is truncated at the level of 4^3, and the bottom solver consists of a significant number of GSRB sweeps. Finally, the solution is interpolated and smoothed upward the V-cycle.

The multigrid solver that we translated [3] employs a 3 dimensional decomposition. Each process keeps a block partition of the problem, which is then further decomposed into multiple 3D blocks that fit well on cache. Fig. 6.4 presents the code organization of the multigrid solver. The program spends the majority of its running time on smooth operations, including GSRB sweeps and boundary exchange among processes. The sweep kernel of the smooth operation has been heavily optimized by the authors [3]. Specifically, all local optimizations such as prefetching, and deep DRAM avoiding [1] were applied efficiently. However, unlike the former optimization, DRAM avoiding results in a major change in the communication pattern. In particular, beside doing a traditional nearest neighbor communication, adjacent processes along the diagonals have to communicate with each other. The effect of this optimization technique is that the number of neighbors that a process communicates with increases from 6 to 26. The resultant communication pattern is shown in Fig. 6.5.

In addition to point-to-point communication, this application also uses an allReduce to compute the maximum error in each timestep and a few Barrier synchronizations points to measure the execution time. However, we did not need to annotate these collective calls. Fig. 6.6 shows how we annotated the smooth function with Bamboo pragma. Note that in the MPI source code the authors inlined this smooth function into the cycleMG procedure.

[1]DRAM avoiding keeps the results of multiple timesteps in the cache without writing back to DRAM

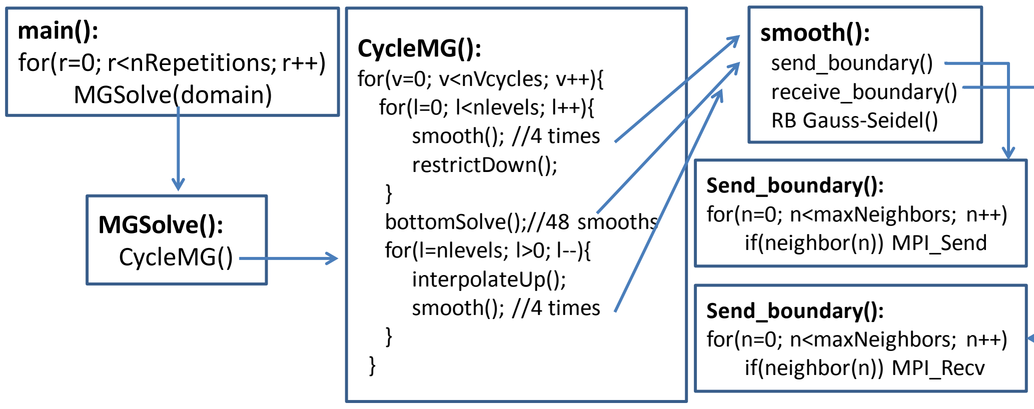


Figure 6.4. MPI code of the multigrid solver. Note that the smooth() function is only for the simplification purpose. In the original code, calls to smooth() are inlined into the cycleMG() function.

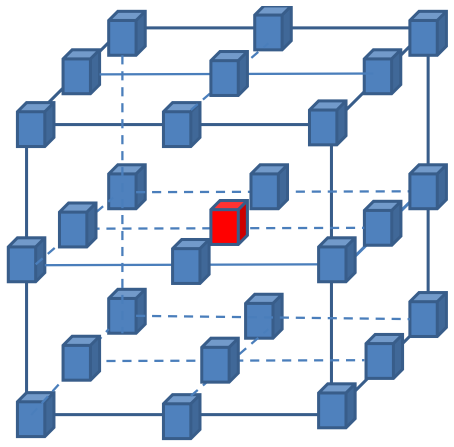


Figure 6.5. A process communicates with 26 neighbors. The process together with its neighbors form a 3D Rubric

```
#pragma bamboo olap
{
  #pragma bamboo send
  send_boundary(&nSends, &sRequests)
  #pragma bamboo receive
  recv_boundary(&nRecvs, &rRequests)
  MPI_Waitall(nSends, sRequests, MPI_ANY_STATUS)
  MPI_Waitall(nRecvs, rRequests, MPI_ANY_STATUS)
  RBGaussSeidel();
}
```

Figure 6.6. Annotating the smooth function with Bamboo pragmas

6.3.3 Performance evaluation

Edison results

We first used Edison to validate the performance of Bamboo. We conducted a weak scaling study, fixing the problem size per processor at 8x128^3 boxes. We configured MPI and Bamboo identically, performing computations on 8 physical cores of each NUMA node and reserving an additional core for communication. Since the MPI variant is a hybrid MPI+OpenMP code, we employed only 2 processes per compute node (1 per socket), each spawning 8 OpenMP threads. Thus, the MPI variant was run with the following command line arguments, where *P* is the total number of cores: *-n P/8 -N 2 -S 1 -d 8*. To run the Bamboo variant, we employed P/8 MPI processes (one per socket), each containing 8 *worker threads*. The command line arguments to run the Bamboo variant is as follows: *-n P/8 -N 2 -S 1 -w 8.* We also performed tests to verify that the virtualization technique employed by Bamboo would not improve the performance via reducing capacity cache misses. We observed that the cache blocking optimization applied by the code's authors eliminated such cache effect.

The left part of Tab. 6.1 shows the execution time of modules of the MPI variant. We can easily calculate that communication accounts for about 20% of the total execution time, and that we have enough available computation to hide communication. While the communication cost grows slightly as the number of cores increases, the execution time for the other activities is quite stable. The right part of Tab. 6.1 shows the relative overhead of communication at each grid level. It can be seen that communication overhead increases by a factor of 2 from the finest grid L0 to L1, slowly increases (L1 to L2 and L2 to L3), or saturates from L3 to the coarsest grids L4.

Fig. 6.7 compares the performance between MPI and Bamboo code variants in a weak scaling study. We can see that both MPI and Bamboo are highly scalable (in a

Table 6.1. Left: execution time in seconds of different modules in the multigrid solver. Right: the relative cost of communication at each grid level (the smaller level, the finer the grid).

Cores	Comm	Compute	pack/unpack	inter-box copy	Comm/total time at each level				
					L0	L1	L2	L3	L4
2048	0.448	1.725	0.384	0.191	12%	21%	36%	48%	48%
4096	0.476	1.722	0.353	0.191	12%	24%	37%	56%	50%
8192	0.570	1.722	0.384	0.191	13%	27%	45%	69%	63%
16384	0.535	1.726	0.386	0.192	12%	30%	48%	53%	49%
32768	0.646	1.714	0.376	0.189	17%	28%	44%	63%	58%

weak sense) and that Bamboo improves the performance by up to 14%. These results are promising, given that overlapping communication with computation on a multigrid solver is challenging due to the following reasons. First, communication is effective at finest grids only as the message size on these grid levels is still significant. In coarser levels, the message size gets smaller and smaller, increasing the overhead of virtualization. In addition, when moving from a fine to a coarser grid computation shrinks by a factor of 8 whereas communication reduces by only a factor of 4, reducing the efficiency of the overlapping technique. Furthermore, the number of messages that each processor has to communicate messages with its 26 neighbors is significant. This increases the processing overhead of the runtime system.

The results in Fig. 6.7 were obtained with an incremental optimization as follows. There are only 2 levels of buffering in the original source code: i) aggregating small messages into a larger one and ii) a potential message buffering done by MPI runtime. Since Bamboo uses active message [2], it issues an extra data copy from application data buffer to the message payload and vice versa. In this application, such data copies are redundant since we can use the payload as an application data buffer. We manually

[2]payload along with header information

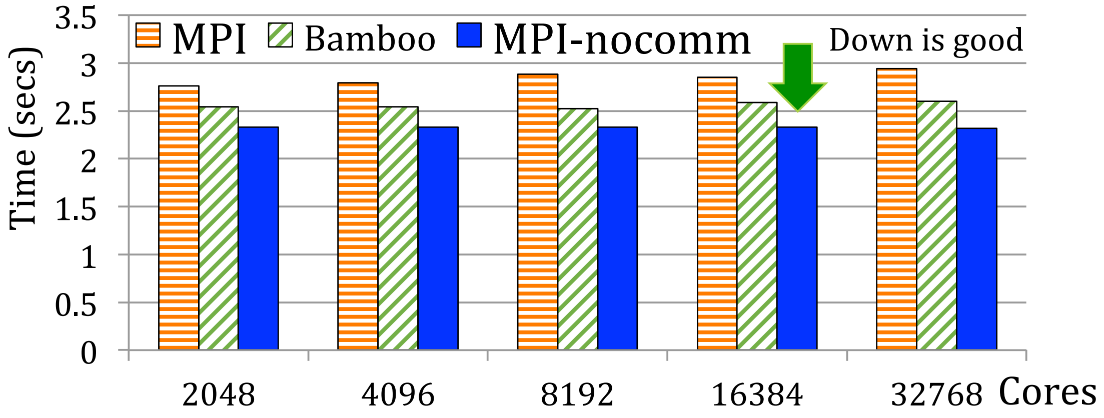


Figure 6.7. Weak scaling study on up to 32,768 processor cores on Edison. At the finest level, each processor accounts for 8x128^3 boxes. Thus, in each V-cycle the finest grid size is 128^3 and the coarsest grid size is 4^3.

perform this optimization and observe that its benefit accounts for up to 3% of the total execution time.

Stampede results

The performance behavior of the multigrid application is complicated due to various degrees of granularity across grid levels. Thus, it is important to make sure that the results can be reproduced on different platforms. To this end, we repeated the experiment on Stampede, a cluster at TACC (Texas Advanced Computing Center). Similar to our experiment on Edison, we also conducted a weak scaling study on Stampede, fixing the problem size per processor at 8x128^3 boxes. We configured MPI and Bamboo identically, performing computations on 7 physical cores of each NUMA node and reserving 1 core for communication. Table 6.2 presents the results of the experimental study. We can see that both MPI and Bamboo are also scalable. Bamboo improves the performance by up to 10%.

Table 6.2. Weak scaling study on up to 16,384 processor cores (2048 Sandy Bridge processors) on Stampede. At the finest level, each processor accounts for 8x128^3 boxes.

Sandy Bridge Procs	T_{MPI}**(s)**	T_{Bamboo}(s)	Speedup
64	2.45	2.24	1.09x
128	2.45	2.24	1.09x
256	2.47	2.29	1.08x
512	2.49	2.30	1.08x
1024	2.56	2.32	1.10x
2048	2.58	2.37	1.09x

6.4 Summary

1. Structure Grid methods employ regular communication and computation patterns to discretize a continuous domain. Based on the grid, solution of partial differential equations can be approximated using numerical computations.

2. We first employed 3D Jacobi, an iterative solver to Poisson's equation that sweeps a 7-point stencil operation across elements of a 3D mesh. We compared Bamboo against many code variants, including the hand-written codes strongly optimized to hide communication overheads. Experimental results on up to 98,304 processor cores on Hopper demonstrate that Bamboo outperforms basic code variants and it meets or exceeds the performance of hand optimized codes.

3. We also used a multigrid solver for Helmholtz equation. Hiding communication overheads by overlapping with computation is challenging as the computation reduces significantly on coarser grids. Nevertheless experimental results show that Bamboo improved the performance of the MPI code variant by 14% on up to 32,768 processor cores on Edison and by 10% on up to 16,384 processor cores on

Stampede.

6.5 Acknowledgements

This chapter, in part, is a reprint of the material as it appears in the article ”Bamboo - Translating MPI applications to a latency-tolerant, data-driven form” by Tan Nguyen, Pietro Cicotti, Eric Bylaska, Dan Quinlan and Scott Baden, which appears in the Proceedings of the 2012 ACM/IEEE conference on Supercomputing (SC’12), Salt Lake City, UT, Nov. 10 -16, 2012.

This chapter, in part, is currently being prepared for submission for publication with Scott Baden. The dissertation author is the primary investigator and author of this material.

Chapter 7

Unstructured grid

7.1 Overview

Unstructured grid is a class of methods for solving problems on complex geometries [84]. Unstructured grid methods employ an arbitrary set of elements to partition a solution domain. Unstructured grid is also useful when employed in the numerical solution of moving boundary problems. Thus, unstructured grid methods have become commonplace in CFD (Computational Fluid Dynamics) [85, 86]. In this chapter, we employ LULESH [28, 87–89], a simplified version of ALE (Arbitrary Lagrangian Eulerian) [90, 91], a method to solve moving boundary, multiple material problems.

7.2 LULESH

LULESH (Livermore Unstructured Lagrange Explicit Shock Hydrodynamics) [28, 87–89] is a proxy application developed by the Co-design project at Lawrence Livermore National Laboratory (LLNL). LULESH employs the Lagrangian hydrodynamics framework to simulate a physical problem governed by 3 conservation equations of mass, momentum, and energy. The latest version of LULESH solves one octant of the spherical Sedov blast wave problem [28] in three dimensions as shown in Fig. 7.1. The Sedov blast wave problem is often used to simulate astrophysical problems such as core-collapse supernovae. This problem has been a standard test for hydrodynamic codes since it can be easily verified by using an analytic solution. In addition, this solution can be scaled to arbitrarily large problem sizes.

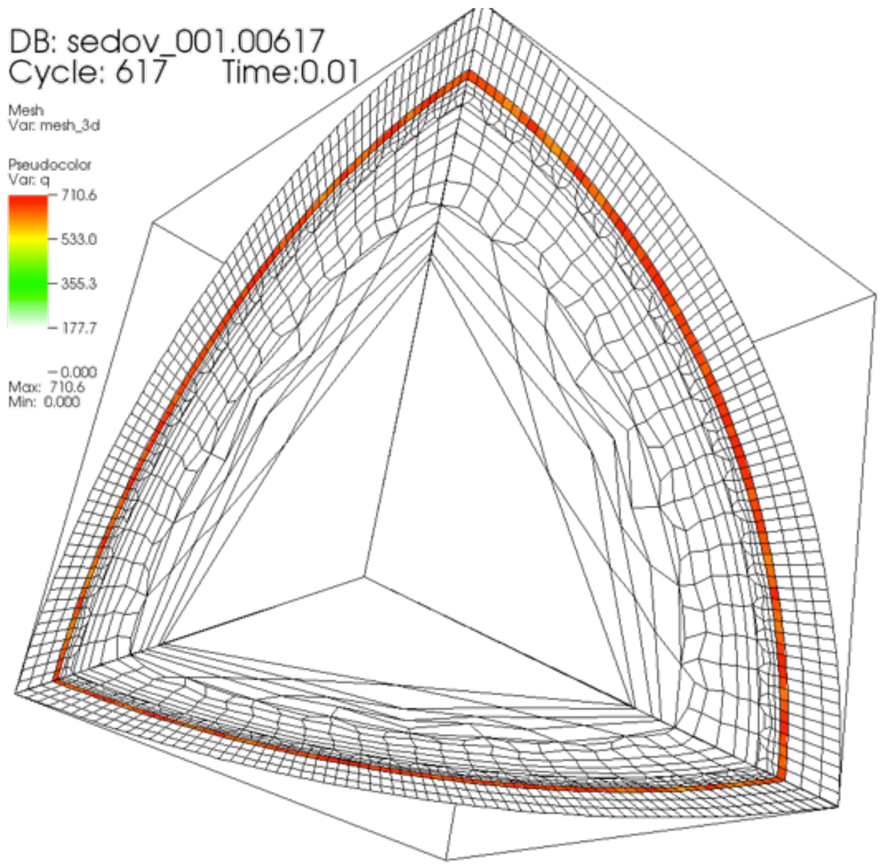


Figure 7.1. LULESH solves one octant of the spherical Sedov blast wave problem using Lagrangian hydrodynamics (Image source [4]).

7.2.1 Staggered 3D partial mesh

The solution domain employed by LULESH is a *staggered mesh*, in which elements are hexahedrons that can be potentially distorted in three-dimensional space. Fig. 7.2 depicts the mesh system. Dependent variables such as ρ, e, and p (i.e. density, internal energy, and pressure) are represented by the element center, whereas kinematic variables such as U, X, and F (i.e. velocity, position, and forces) are represented by element nodes. In Sec. 7.2.2, we will describe how these gradients can be approximated using finite elements. Tab. 7.1 lists and describes all variables that will be used in the remainder of the chapter.

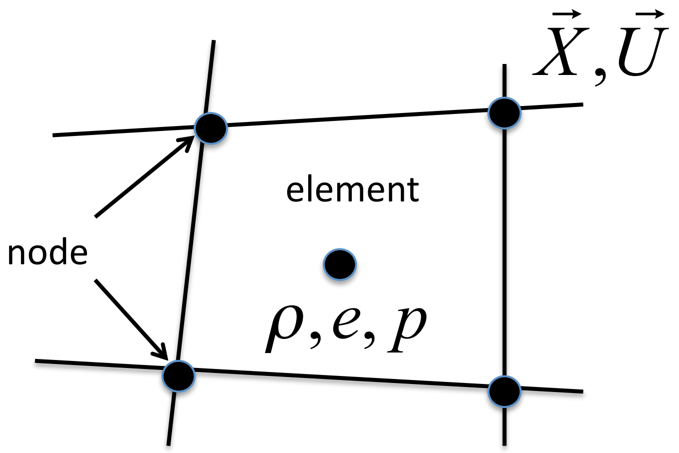


Figure 7.2. Elements of a *staggered mesh* are hexahedrons that can be potentially distorted in three-dimensional space, each consisting of center and nodes.

Table 7.1. Node variables and element variables

Nodal variables	description	element variables	description
$\vec{X}$ = (x, y, z)	position vector	p	pressure
$\vec{U} = (U_x, U_y, U_z)$	velocity vector	e	internal energy
$\vec{A} = (A_x, A_y, A_z)$	acceleration vector	q	artificial viscosity
$\vec{F} = (F_x, F_y, F_z)$	force vector	V	relative volume
m_0	nodal mass	l_{char}	characteristic length
		ε	diagonal terms of deviatoric strain

In order to simulate a problem with various materials, LULESH introduces the notion of *region*. Each *region* is a subset of the mesh, and it simulates a material. For the sake of simplicity, LULESH currently employs the same material (ideal gas) for all *regions*, but varies the size of the *region*. In addition, LULESH provides additional costs for some *regions* to simulate the difference in computational costs associated with material properties. Although the current simulation setup is sufficient to represent a practical multi-material problem, the limitation of repeating a single material may be lifted in later versions of LULESH.

7.2.2 Lagrange leapfrog algorithm

The solution of the Sedov blast wave problem is approximated by discretizing the time domain into time steps. The discrete time increment $\Delta T = t^{n+1} - t^n$ is not a fixed value but is re-calculated at every time step. The solution is updated from t^n to t^{n+1} using the *Lagrange leapfrog* algorithm [89], which consists of 3 phases to update the following variables: nodal variables, center variables, and the time increment.

Advance nodal variables

The first phase of the *Lagrange leapfrog* algorithm updates nodal variables. Initially, these updates are conducted locally, containing the following steps.

1. *Calculate nodal forces*: this is the most compute-intensive kernel of the code. A volume force contribution is calculated within each mesh element by integrating the volumetric stress contributions. The force in each element is used to distribute a force contribution to each of its surrounding nodes.

2. *Calculate nodal accelerations*: the acceleration vector is calculated from the force by simply applying Newton's second law: $\vec{F} = m\vec{A}$.

3. *Apply acceleration boundary* conditions as needed.

4. *Updated nodal velocities*: this module integrates the acceleration at each node to advance the velocity at the node $\vec{U}^{n+1} = \vec{U}^n + \vec{A}\Delta t^n$. If the resulting velocity is

smaller than a cut-off value, it will be set to zero.

5. *Update nodal positions* by integrating nodal velocities: $\vec{X}^{n+1} = \vec{X}^n + \vec{U}^{n+1}\Delta t$.

Force, velocity, and position variables on the surfaces of the solution domain are then exchanged between neighbor processes. MPI point-to-point communication is used, and each process communicates with up to 26 surrounding neighbors. Fig. 7.3 shows the the code for exchanging the force variable between each process and its 26 neighbors. The codes for exchanging velocity and position variables are similar.

```
void CommRecv(Domain* domain,...)
{
  for(each source in 26 neighbors) //6 planes, 12 edges, and 8 vertices
    if(sharedPlane || !sharedPlane && sharedEdge || !sharedPlane && !sharedEdge &&
        sharedVertices){
      MPI_Irecv(domain, ...,source, ...);
    }
}
void CommSend(Domain* domain,...)
{
  for(each destination in 26 neighbors) //6 planes, 12 edges, and 8 vertices
    if(sharedPlane || !sharedPlane && sharedEdge || !sharedPlane && !sharedEdge &&
        sharedVertices){
      MPI_Isend(domain, ...,destination...);
    }
}
void CalcForceForNodes(Domain* domain)
{
  Index_t numNode = domain->numNode();

  MPI_Request request[52];
  for (Index_t i=0; i<52; ++i) {
     request[i] = MPI_REQUEST_NULL;
  }
  #pragma bamboo olap layout cubeRubikConnector
  {
    #pragma bamboo receive
      CommRecv(domain, ...);
    #pragma bamboo send
      CommSend(domain, ...);
    MPI_Waitall(52, request, MPI_STATUS_IGNORE);
  }
}
```

Figure 7.3. Each process communicates with up to 26 surrounding neighbors to exchange forces. Codes for exchanging position and velocity are similar.

Advance center variables

Center variables are advanced using updated nodal variables.

1. *Calculate kinematic element quantities*: this step calculates terms in the total strain rate tensor ε, which are used to compute the terms in the deviatoric strain rate tensor.

2. *Calculate artificial viscosity*: this step calculates the artificial viscosity term q for each element. For algorithm details and the mathematical aspect of the algorithm, see [92].

3. *Apply material properties*: this step updates the pressure p and internal energy e. The sound speed is then calculated based on p and e. The sound speed is useful in computing time increment of the next time step.

4. *Update element volumes*: this step simply updates current relative volume V^n to the new volume V^{n+1}.

Since these computations do not make use of remote data, this phase does not contain MPI code.

Update the time increment

At every time step, the time increment is recalculated. However, the calculation only applies to elements whose volume is changing. Courant and Hydro constraints are used. The Courant constraint is calculated via dividing characteristic length for the element by its change in volume. Hydro constraint is the maximum allowable volume change divided by the change in volume. The new time increment is then calculated from these constraints. It is important to find the minimum value of the time increment across all processes. Thus, MPI_Allreduce is used to perform this task as shown in Fig. 7.4.

```
void TimeIncrement(Domain* domain){
   CalcTimeConstraintsForElems(domain); //calculate the Courant and Hydro
       constraints
   Real_t dt = 1.0e+20;
   if (domain->dtcourant() < dt) {
      dt = domain->dtcourant() / 2.0;
   }
   if (domain->dthydro() < dt) {
      dt = domain->dthydro() * 2.0 / 3.0;
   }
   MPI_Allreduce(&dt, &newdt, 1, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);
   domain->deltatime() = newdt ;
}
```

Figure 7.4. MPI code to update the time increment

7.3 Performance evaluation

7.3.1 Hopper results

We first conducted a weak scaling study on Hopper, where we maintained the number of data elements per process at 92^3. We evaluated 2 code variants: MPI is the original code developed by the Co-design project at LLNL and Bamboo is the task graph code generated by Bamboo. Throughout the experiment, we employed a cubic number of MPI processes as required by the provided MPI code. All jobs were launched using the *aprun* command. The MPI variant was run with the following command line arguments: *-n P -N 16 -S 4*, where *P* is the total number of cores. Bamboo employed only 4 MPI processes per compute node. Thus, it was run with the following command line arguments: *-n P/4 -N 4 -S 1*. For LULESH, we used the default programming environment (PGI) as it delivered the best performance.

Fig. 7.5 shows the results of the MPI and Bamboo variants on up to 32,768 processor cores. It can be seen that the execution time of the MPI variant grows as the number of processor cores increases, though ideally this curve would be flat as we maintain the same problem size per process. This upward trend in execution time can be well explained by the growth in communication delays as more processor cores are used.

Bamboo performance, however, is well maintained as a nearly flat rate over a wide range of core counts (i.e. from 64 to 32,768). This result demonstrates the ability of Bamboo to tolerate communication delays.

For all core counts, Bamboo used a fixed virtualization factor. In particular, the number of tasks per worker thread (i.e. MPI process in respect to the MPI code variant) was always 8. Since we must use a cubic number of tasks, the virtualization factor can be 1, 8, 64, 512, and so on. We found 8 to be optimal as it not only enables communication overlap but also minimizes the overhead of over-decomposing the problem.

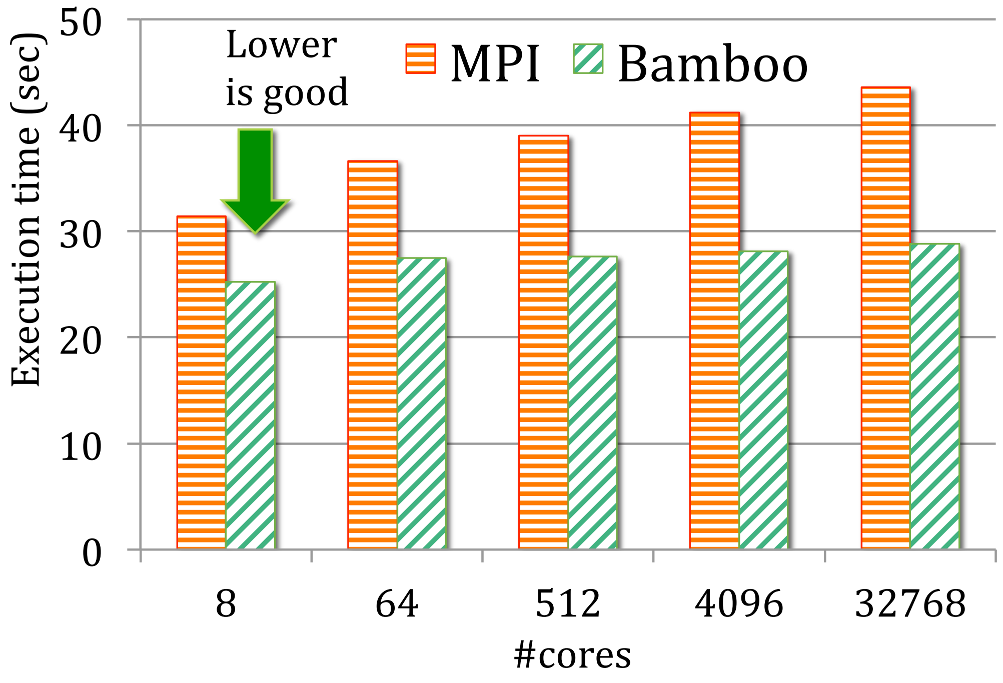


Figure 7.5. Weak scaling results on Hopper. Local domain per process: 92^3 and number of iterations: 10.

Though the MPI code variant can't run with an arbitrary number of MPI processes, this is not a constraint with Bamboo. Due to the execution model based on dynamic scheduling, there is no restriction in the number of MPI processes nor the process geometry. Rather, as long as Bamboo spawns a cubic number of tasks, these tasks will be mapped and executed by MPI processes. This capability enabled us to quickly design a strong scaling study, as shown in Fig. 7.6. In this study, we fixed the problem size at 1472^3 while varying the core count from 4,096 to 32,768. Since the number of processor cores is not always a perfect cube, the virtualization factor varies. Specifically, the virtualization factor was set at 8, 4, 16, and 8 (tasks per MPI process) on 4K, 8K, 16K, and 32K processor cores, respectively. It can be seen in Fig. 7.6 that Bamboo realizes linear speedup. The importance of this result is that strong scaling is commonly used in practice and hiding latency in strong scaling is very challenging as the computation shrinks steadily.

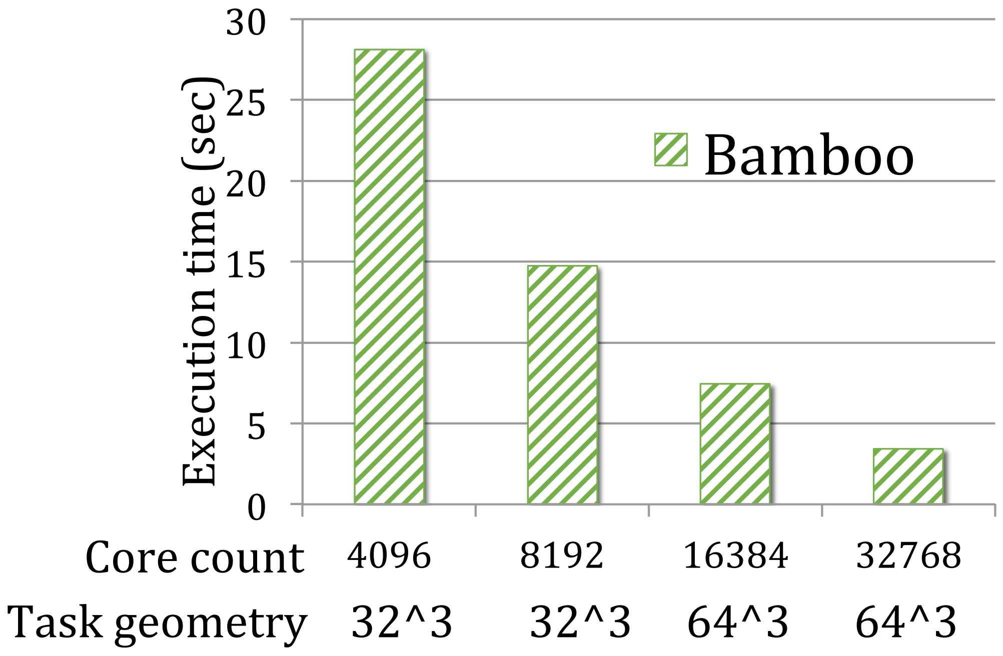


Figure 7.6. A strong scaling study on Hopper. We fixed the problem size at 1472^3. Bamboo allows LULESH to be run on an arbitrary number of processes.

Another reason for the good results shown in Figures 7.5 and 7.6 is load balancing. In particular, we configured the runtime system with 1 MPI process per NUMA node, each running 4 worker threads (1 worker thread per processor core). Worker threads within an MPI process do not maintain their own task queues. Rather, they share a common task queue so that computations can be more evenly assigned to worker threads.

For example, with a virtualization factor of 16 there are 16x4=64 tasks per MPI process executed by 4 worker threads. These 4 worker threads will dynamically pull tasks from the shared queue as long as they are not occupied and the task queue is not empty. The dynamic task scheduling scheme and the single task queue configuration make sure that the computations assigned to a NUMA node can be evenly divided into processor cores and these cores never become idle unless the shared queue runs out of tasks.

7.3.2 Edison results

We also conducted a strong scaling study on Edison. We fixed the problem size at 896^3. As on Hopper, all jobs on Edison were launched using the *aprun* command. The MPI variant was run with the following command line arguments: *-n P -N 16 -S 8*, where *P* is the total number of cores. The MPI variant applied only when a cubical number of processes was used. Bamboo employed only 2 MPI processes per compute node. Thus, it was run with the following command line arguments: *-n P/8 -N 2 -S 1*.

Fig. 7.7 presents the performance of the MPI and Bamboo variants on up to 16,384 processor cores. We observed that Bamboo ran faster than MPI, though the performance improvement was a little less significant than that on Hopper. On Edison, the Bamboo execution model based on dynamic scheduling also worked well, enabling Bamboo to run with an arbitrary number of MPI processes. It can be easily seen that the execution time of the Bamboo variant was steadily reduced by a half every time we doubled the number of cores.

7.4 Summary

1. This chapter presented results on LULESH, a proxy application of hydrodynamics computation, which is an unstructured grid method. The application solves one octant of the spherical Sedov blast wave problem in three dimensions.

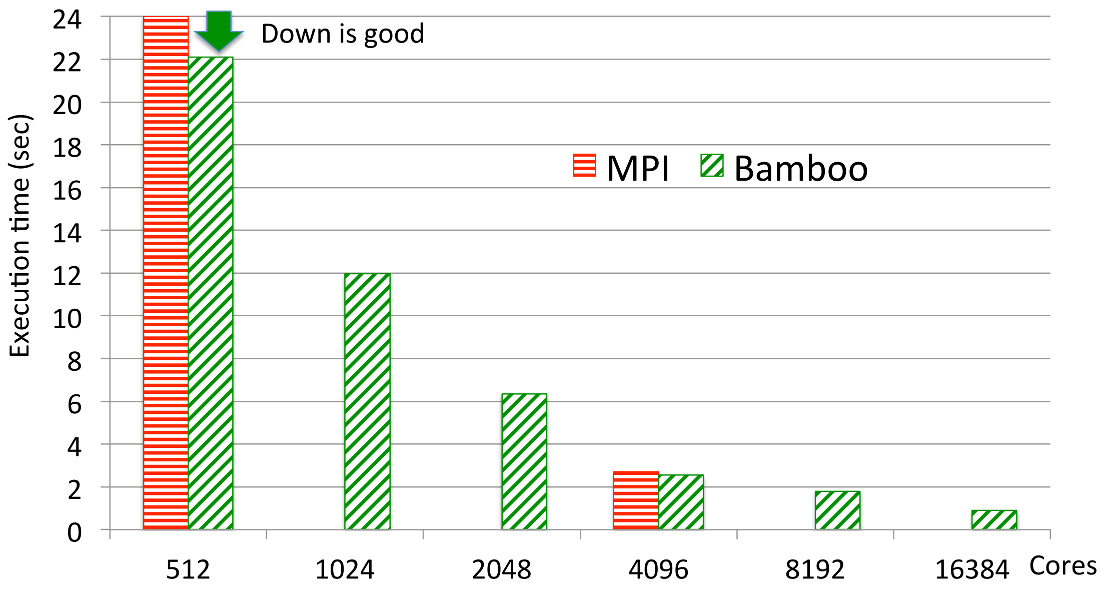


Figure 7.7. Strong scaling results on Edison. Problem size: 895^3 and number of iterations: 10.

2. Experimental results on up to 32,768 processor cores of Hopper demonstrate that Bamboo is able to hide communication overheads, significantly increasing the performance of the original MPI code. The effect is that the application can tolerate communication delays and thus it becomes more scalable.

3. In addition to the performance advantage, Bamboo also demonstrates its flexibility in configuring the process grid. In particular, the task graph execution model and the dynamic task scheduling support allow processor cores to execute tasks based on the availability of data. Thus, Bamboo is able to run the LULESH application on any arbitrary number of processor cores, whereas the MPI input code requires a perfect cube of MPI processes.

7.5 Acknowledgements

This chapter, in part, is currently being prepared for submission for publication with Scott Baden. The dissertation author is the primary investigator and author of this material.

Chapter 8

Advanced node technologies

8.1 Overview

At present, it appears that further improvements to HPC systems will mainly come from enhancements at the node level [8–10]. Node architectures are changing rapidly, and a heterogeneous design that uses a *device* (i.e. coprocessor or accelerator) to amplify node performance is gaining traction. However, heterogeneous nodes challenge the application programmer as follows. First, performance amplification significantly raises communication costs relative to computation. Thus, we are required to tolerate communication delays [23, 25, 53, 58, 62, 93–102], avoid them [71, 103–105], or both. Second, processor mapping is challenging due to the heterogeneous design. This task also requires significant application redesign to (1) work within the limitations of the interface to the memory subsystems and (2) to use the interface in a way that utilizes the resources efficiently. Third, the performance differential between *devices* and the controlling processors introduces the need to solve a load-balancing problem within the node. This is true even if the application has no inherent load-balancing problem. In this chapter, we discuss the support of Bamboo on state-of-the-art computing platforms employing advanced node technologies. We consider two emerging device architectures: Graphics Processing Units (GPUs) and Many Integrated Core (MIC).

8.2 Graphics Processing Unit

8.2.1 The GPU architecture

Graphics Processing Unit (GPU) has become a powerful means of accelerating compute-intensive and bandwidth-intensive applications. Each GPU comprises many simple processor cores AKA *stream processors*, each consisting fewer functional units per core and operating at a lower clock rate compared to conventional single-core and multi-core processors. These *stream processors* are organized into vector processors AKA *stream multiprocessors* (SMs). Within a single SM, *stream processors* execute SIMD instructions (single instruction mutiple data) in lockstep.

In this dissertation, we use NVIDIA K20 GPUs, a product of the Kepler architecture [106]. Each K20 GPU consists of 2496 simple processor cores running at 706 MHz. With the Kepler architecture, NVIDIA first introduces *stream multiprocessors extreme* (SMX), an extension to the original SM architecture, which allows multiple groups of SIMD instructions to execute at a time. The cores of a K20 GPU are organized into 13 SMXes. Each SMX contains 64KB of register and a 64KB configurable scratchpad memory, which can be partitioned into L1 cache and a software-managed memory called *shared memory*. L2 cache, however, is shared among SMXes and may not be configured. A K20 GPU contains 5GB of device DRAM with a 208 GB/s of memory bandwidth.

8.2.2 CUDA and MPI+CUDA programming models

CUDA (Compute Unified Device Architecture) is a well-known parallel architecture and programming model developed by NVIDIA. Under the CUDA programming model, a program executes sequences of kernels, functions that run under the Single Instruction Multiple Threads (SIMT) model. Each CUDA kernel runs a set of threads, which are hierarchically organized into three-dimensional *thread blocks*. The programming model conceptually partitions these *thread blocks* into a two-dimensional grid. CUDA dynamically assigns each thread block to a single *stream multiprocessor*. A thread block is further broken down into a collection of multiple *warps*, each a group of scalar threads that execute in SIMD fashion.

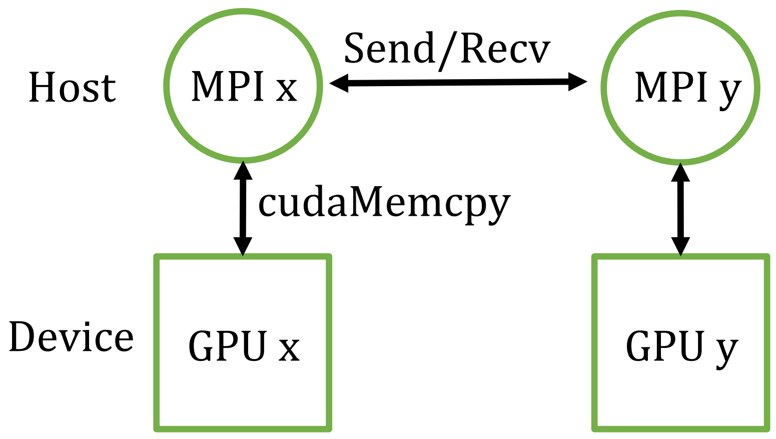


Figure 8.1. The traditional MPI+CUDA programming model. Each GPU is a *device* attached to a CPU called *host*. *Devices* communicate with each other via their *hosts*.

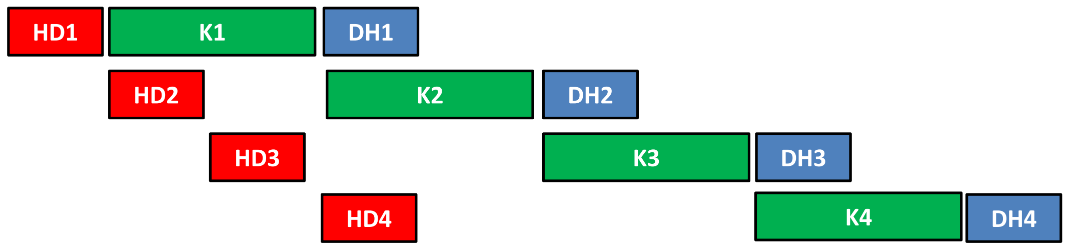


Figure 8.2. Overlapping GPU kernels with host-to-device and device-to-host data transfers. HD: host-to-device, K: kernel, DH: device-to-host.

MPI+CUDA is a hybrid programming model commonly used to parallelize application across multiple GPUs. Under this model, each GPU works as a *device* attached to a CPU called *host*, as shown in Fig. 8.1. *Devices* communicate with each other via their *hosts*, requiring the programmer to use CUDA to program the data transfer between *host* and *device*. There are two challenges associated with the MPI+CUDA programming model. First, MPI only handles the communication among *hosts*. Thus, any attempt to optimize the hybrid code must treat MPI and CUDA components separately. Second, hiding communication overheads is challenging. Indeed, Fig. 8.2 shows that the programmer needs to pipeline the GPU kernel and data transfer to hide the communication delays between *host* and *device*. It becomes even more complicated when the programmer orchestrates host-host and host-device to overlap all communication delays arising in an MPI+CUDA program.

To demonstrate that developing communication-tolerant code under the traditional MPI+CUDA programming model is non-trivial, we used *3D Jacobi* as a motivating example. The basic communication code of *3D Jacobi* is similar to the one that has been shown previously in Fig. 3.8, except for one additional step. That is, we added copying instructions to explicitly transfer data between *host* and *device* before and after launching the kernel that performs the mesh update on GPU. Based on this implementation, we manually reformulated the application's data structures and embedded a scheduling algorithm to obtain a highly optimized code, which is able to hide both host-host communication and host-device data transfer. In particular, we first over-decomposed the data partition assigned to each GPU into multiple disjoint 3D blocks. The top row of Fig. 8.3 describes how the application's data is distributed to GPUs under a regular block decomposition scheme. The bottom row of Fig. 8.3 shows how each partition is over-decomposed into smaller 3D blocks. We then statically scheduled computation and communication with the algorithm shown in Fig. 8.4, so that Host-Host communication cost is hidden by GPU computation and Host-Device transfer is pipelined with the Host-Host communication.

8.2.3 GPU-aware MPI

Though we have no doubt that the optimization technique presented above can improve performance, its implementation is time consuming and error-prone. In addition, it complicates the original code, entangling further software development. Thus, one may

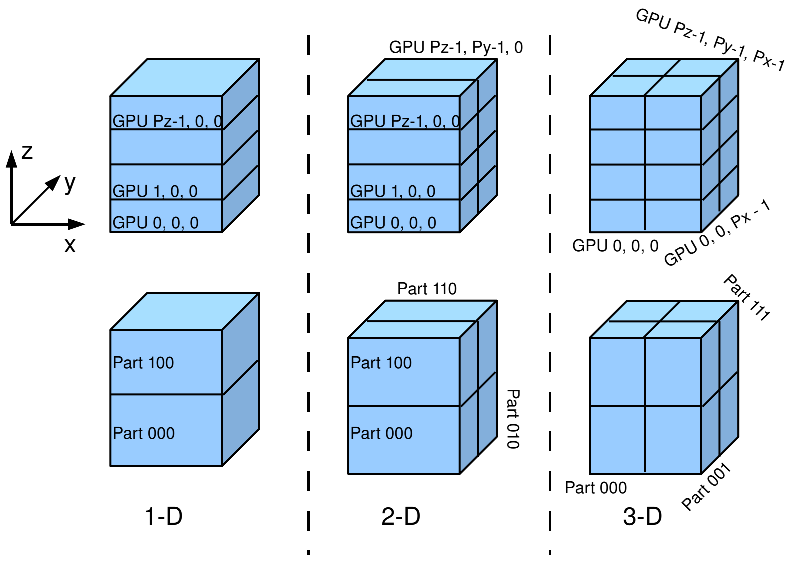


Figure 8.3. 1, 2 and 3-D decomposition. a)Top: Problem size is distributed over GPUs indexed in 3-D Cartesian coordinates b)Bottom: Subproblem at each GPU is divided into smaller subsubproblems to increase the opportunity to hide latency.

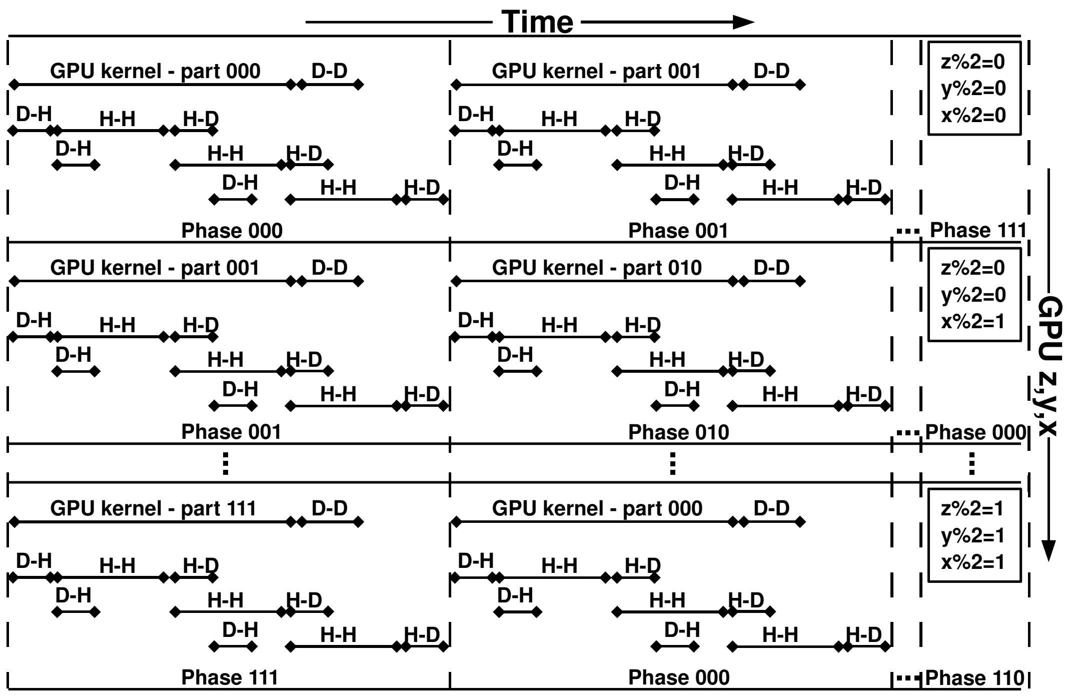


Figure 8.4. Host-Host Latency hiding and Host-Device pipelining. With 1-D decomposition, only phases 000 and 100 are used. With 2-D decomposition, phases 000, 010, 100 and 110 are used. All phases are used in 3-D decomposition.

think of translating MPI+CUDA code into its task dependency graph form by using the Bamboo translator. Unfortunately, the resulting program can tolerate the communication among *hosts* only. The reason is that the CUDA interface does not have any connection with MPI. Thus, the data transfer between *host* and *device* can't be factored out of the task execution on the *device*.

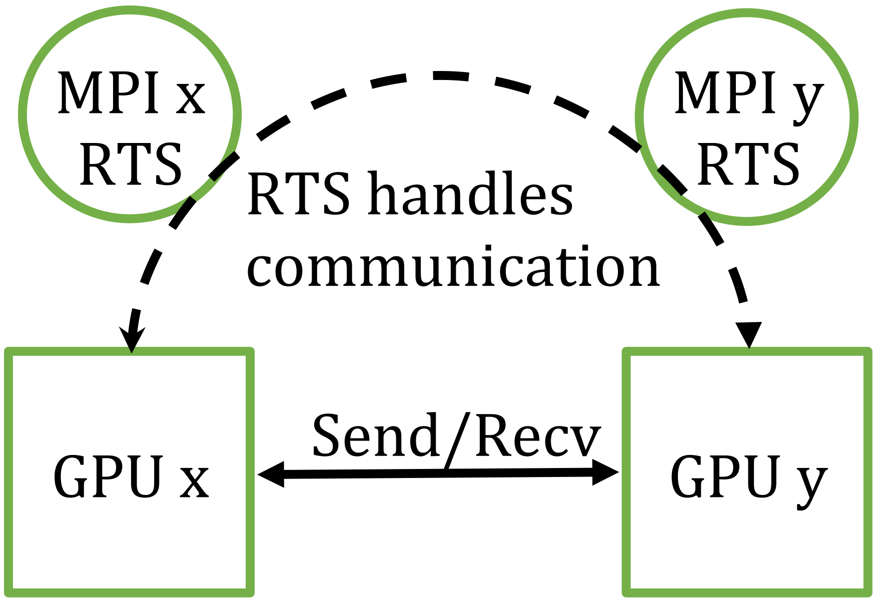


Figure 8.5. A new programming model allowing devices to exchange message directly

We propose GPU-aware MPI, a new programming model that allows MPI communication routines to take device memory as buffer for sending and receiving data. Fig. 8.5 presents this model. With GPU-aware MPI, the programmer can manage the communication between *devices* without the need to explicitly route data via the *hosts*. Instead, it is the responsibility of the compiler and runtime system to handle the data transfer between *host* and *device*. Our proposal is similar to those proposed by MPI-ACC [107, 108] and MVAPICH2-GPU [109]. However, we integrated GPU-aware MPI with Bamboo and the Tarragon extension proposed in Chapter 3. The result is that we can turn MPI+CUDA program into a proper task dependency graph form, where host-device transfer is factored out of task and is represented as edge of the graph. Thus, we can automatically mask both host-host and host-device communication overheads. MPI-ACC and MVAPICH2-GPU, on the contrary, cannot automatically hide communication overheads.

Another advantage of the GPU-aware programming model compared to the traditional one is that it helps isolate the application from technological changes in interconnection network. For example, GPUDirect is a promising interconnect technology that is currently under development and deployment [110, 111]. GPUDirect enables GPUs to communicate with each other directly without coordinating with hosts. In order to take advantage of GPUDirect, however, applications developed under the traditional MPI+CUDA programming model must be significantly rewritten. In addition, with GPUDirect there are at least 2 paths from a GPU to another: a direct path connecting 2 GPUs and an indirect path via the hosts. A static routing approach is obviously not suitable for this scenario. Due the GPU-aware programming model, communication between devices and host-device data transfer can be factored out of the task execution. As a result, the runtime system can decide on the fly the best path for a message based on the source/destination information and the current traffic conditions on the compute node and the network.

8.2.4 Performance evaluation

We evaluated the new programming model using the *3D Jacobi* solver. We compared 5 code variants. The first and second variants, *MPI-basic* and *MPI-olap*, employ the traditional MPI+CUDA programming model. The third variant, *Bamboo*, is the task graph program obtained by translating *MPI-basic*. The fourth variant, *Bamboo-GPU*, is generated by the Bamboo translator from a basic MPI+CUDA code written under the GPU-aware programming model. The fifth variant, *MPI-nocomm*, was obtained by removing all host-host and host-device communication calls.

We conducted a weak scaling study on Stampede. This cluster consists of a few GPU queues, each containing up to 32 GPU nodes. Each GPU node is equipped with one K20 GPU and two 8-core Sandy Bridge processors. GPU nodes are connected by a Mellanox FDR InfiniBand interconnect. We evaluated all code variants with the base problem size of 510x512x128 per GPU. We ran experiments on up to 32 GPUs. Due to the small scale, we employed a 1D decomposition scheme.

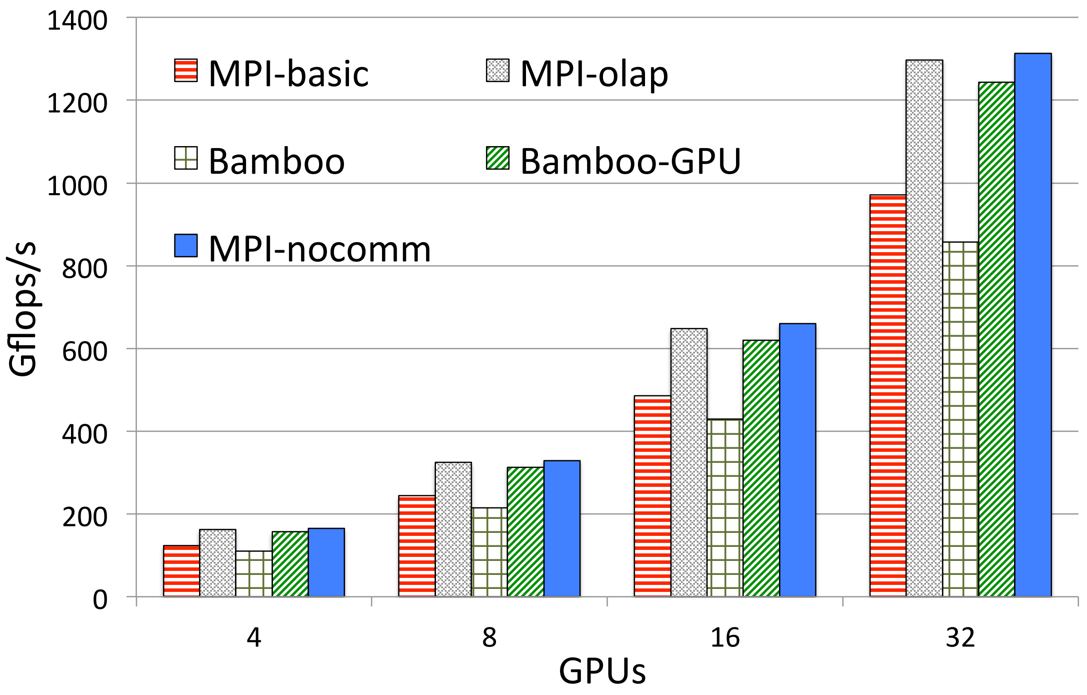


Figure 8.6. A weak scaling study on 3D Jacobi. We use up to 16 GPUs on Stampede. Bamboo outperforms MPI-basic, though it runs a bit slower than the hand optimized code.

Fig. 8.6 shows the performance in GFLOP/s of all code variants. It can be seen that *Bamboo-GPU* and *MPI-olap* significantly outperform *Bamboo* and *MPI-basic*. The reason is that *Bamboo-GPU* and *MPI-olap* can overlap both host-host and host-device communication with the computation. Although *Bamboo* can overlap host-host communication, host-device is not handled properly, significantly slowing down the performance. In particular, even when tasks share the same *device*, they must communicate via their common *host*, creating unnecessary traffic on the PCI Express bus. It is the result of not factoring host-device communication out of the computation.

We attribute the performance improvements of *Bamboo-GPU* compared to *Bamboo* and *MPI-basic* to the following optimizations.

- The knowledge of host-device transfer enables *Bamboo-GPU* to take advantage of locality, such as tasks computing on the same GPU only exchange the header information of messages. This optimization can save significant bandwidth of the PCI Express bus connecting *host* and *device*. We found that this optimization is very significant at small scales, where the bandwidth between host and device is more critical than between hosts.
- Under the GPU-aware programming model, the runtime system has to optimize host-device transfer. We reimplemented the runtime system so that it becomes GPU-aware. Specifically, the new implementation of Tarragon employs pinned memory to buffer messages. Using pinned memory can significantly increase the bandwidth between *host* and *device* [112, 113]. Indeed, Volkov and Demmel found that copying data at non-pinned rate realized only a half of the peak sustained bandwidth [112].
- We used asynchronous memory copy variants to avoid implicit synchronization on the GPU. We will discuss the effect of synchronous routines in Sec. 8.2.5.
- We pre-allocated messages, which includes both host memory and device memory buffers, and recycled them during the program execution. Recycling memory is important because allocating and deallocating device memory are not only costly but also block the whole GPU, reducing overlap.

Compared to MPI-olap, the performance of Bamboo is a little lower (about 4%).

The reason for this slight performance reduction is that task has to buffer out-going data and the runtime system has to buffer incoming data. The cost for buffering data is often small, since a GPU often has remarkably high DRAM bandwidth. Another reason is that we conducted the performance study at a small scale, where the effects of network congestion and processor performance variation are very modest. Thus, the benefits of using a dynamic scheduling scheme are outweighed by its overheads.

8.2.5 Notes to the programmer

CUDA employs 3 different engines to service kernel calls and data transfer requests (one for host-to-device and another for the reverse direction). These engines run in parallel, except for the case that a special CUDA routine blocks the entire GPU. For example, *cudaDeviceSynchronize()* blocks the GPU until all previous activities have completed. To realize the expected communication overlap, the programmer should avoid using CUDA routines that causes such a total synchronization on the GPU. Instead, the programmer should use similar variants that block only the caller thread. For example, *cudaStreamSynchronize()* can be used to replace *cudaDeviceSynchronize()*. We list special CUDA routines that the programmer should avoid in Appendix D.

8.3 Many Integrated Core

8.3.1 The MIC architecture

Many Integrated Core (MIC) is a state-of-the-art many-core processor architecture designed and developed by Intel. Intel refers to it as a co-processor. The MIC architecture is based on simple, in-order, x86-like processor cores running at low clock speeds. It supports legacy x86 codes, making application migration easier than on accelerators that support vastly different ISA.The initial offering of the MIC architecture is the Intel Xeon Phi with the code name *Knights Corner* (KNC). This coprocessor consists of 61 cores, each an Intel Pentium-like processor (2 pipe in-order superscalar design). However, each core of KNC includes a 512-bit SIMD ALU that can perform 8 double-precision floating-point operations per clock cycle. Processor cores communicate, and access on-chip DRAM, via a 512-bit wide, bi-directional ring bus and 8 memory controllers. To reduce the gap between register and memory, each core contains 32 KB of private L1 and a partitioned 512KB L2.

Since MIC is a shared-memory architecture with many processor cores crammed on a chip, it supports both multithreading (Pthreads, OpenMP, Intel Threading Building Blocks (TBB), and Cilk) and message passing (MPI only) programming models. A hybrid approach is also supported, where multiple processes, each spawning multiple threads, run on a single MIC. To hide latency due to data and instruction fetching, MIC supports up to 4 *hardware threads* (AKA contexts) per core. The Intel documentation states that a minimum of 2 contexts are required to maximize performance. Because MIC supports familiar programming models, it provides a friendly programming environment and high opportunities to port legacy code to accelerate the performance.

8.3.2 Execution modes on a MIC cluster

To validate the performance of Bamboo on the MIC architecture, we used Stampede [77], one of the largest machines based on MIC built to date located at the Texas Advanced Computing Center (TACC). This cluster consists of 6400 compute nodes, each equipped with 1 Intel Xeon Phi XE10P coprocessor (KNC) and 2 Intel Xeon E5 8-core processors (Sandy Bridge). Each compute node also contains 32GB (4 x 8GB DDR3) of host memory (NUMA) and 8GB of on-board DDR5 device memory. The coprocessor is connected with the host via a PCIe connection. Nodes communicate via a Mellanox FDR InfiniBand interconnect with a 2-level fat-tree topology. Stampede supports many compilers to build and run codes on hosts. However, it is currently mandatory to use

Intel's compilers (icc and IMPI) to compile and execute codes on MICs. We used the Intel C++ Composer XE 2013 suite (version 13.0).

Stampede provides programmers with 5 execution modes: *host-host*, *MIC-MIC*, *symmetric*, *offload*, and *reverse offload*. This dissertation considers the first three modes. *Host-host* mode executes all computations on the hosts and doesn't use the MICs at all. In *MIC-MIC* mode, the program allocates data and performs computations locally on MICs. *Symmetric* mode supports heterogeneous computing by considering MICs and hosts as peers in one big SMP (symmetric multiprocessor) node. This dissertation does not use either *offload* or *reverse offload* mode where one resource (device or host, respectively) migrates computation to the other. This mode similar to how processors accelerate currently, though accelerators support only *offload* mode.

8.3.3 Performance evaluation

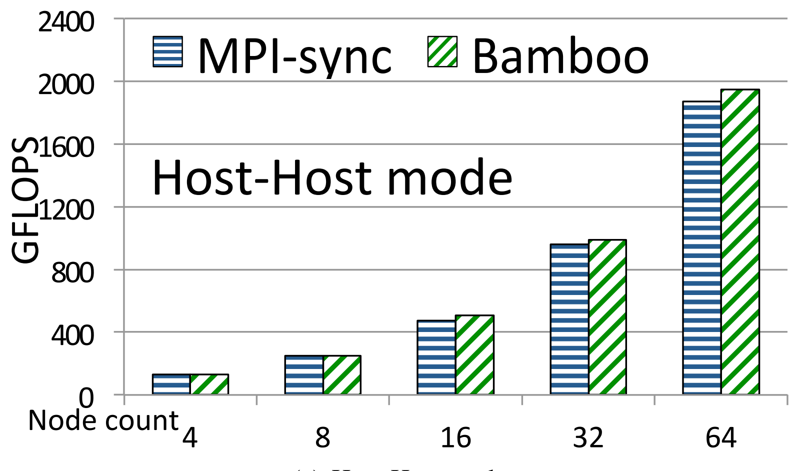
We used *3D Jacobi* to evaluate Bamboo on MIC. We employed the MPI+OpenMP programming model. The basic MPI code of *3D Jacobi* has been shown previously in Fig. 3.8. We used OpenMP to parallelize the workload assigned to each processor/-coprocessor. Unlike GPU, with MIC we are able to port the runtime sytem on both the coprocessor and the host. We configured Tarragon as follows. We initialized MPI processes with the THREAD_FUNNELLED mode. Under this mode, each process consists of multiple threads, but only the main thread can handle communication by making MPI calls. We then bound the main thread (AKA *message handler* thread) to core #0 on MIC using sched_setaffinity(). We then created a set of 236 *hardware threads* and map them to cores #1 to #59 in a block fashion (4 consecutive threads on each core). We next created a worker thread for Tarragon and bound this thread to the set of 236 *hardware threads*. This worker thread runs one task at a time by mapping OpenMP threads spawned in the task to *hardware threads*. *Message handler* and *worker* threads run until the program completes. OpenMP threads may or may not run to completion, depending on the OpenMP implementation. We did not use core #60 to eliminate noise due to the microOS of MIC. On host, there are 16 cores indexed from 0 to 15. As a result, we created 2 MPI processes per host. We mapped the *message handler* threads of these processes to cores #0 and #8, and 7 worker threads to cores #1 to #7 and #9 to #15, respectively. All these setups are totally independent of the application program.

Weak scaling study

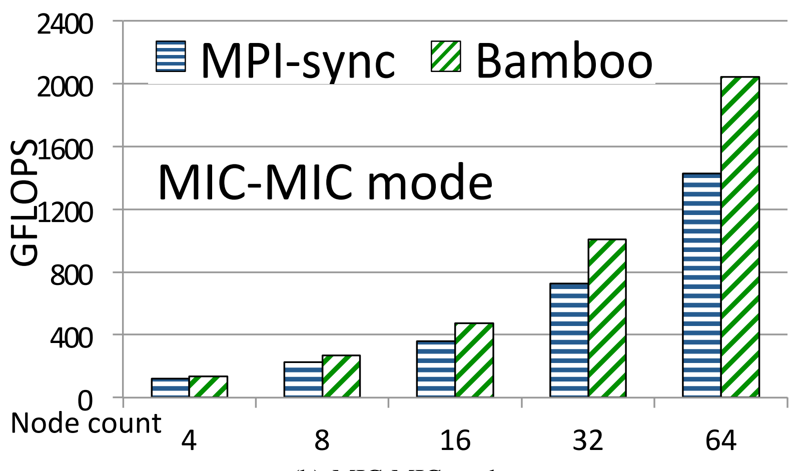
We first conducted a weak scaling study, fixing the problem size at 384x384x384 per node. Thus, we increased the problem size in proportional to the number of nodes. Fig. 8.7 presents the results of *3D Jacobi* with different modes on up to 64 nodes. Consider the first 3 subfigures (8.7(a), 8.7(b), and 8.7(c)), we can see that MIC-MIC performs well on 4 and 8 nodes but it does not scale as well as the other 2 modes. This can be explained by looking at Fig. 8.7(d), where communication cost in the *Host-Host* mode is much smaller than it is in the *MIC-MIC* mode. The *symmetric* mode provides both impressive performance and scalability. By splitting workload across two Sandy Bridge processors, we can take advantage of Xeon E5 processors and reduce the amount of off-node communication. Results shown in Fig. 8.7(d) indicate that the percentage of communication in the *symmetric* mode is twice as large as it is in the *Host-Host* mode. However, the absolute time for communication in two modes is the same as the *symmetric* mode runs twice as fast as the *Host-Host* mode.

We note that the performance benefit of using Bamboo on only a small number nodes is modest in Host-Host mode. Yet, we still recover about half the communication costs as we did when we used MIC.

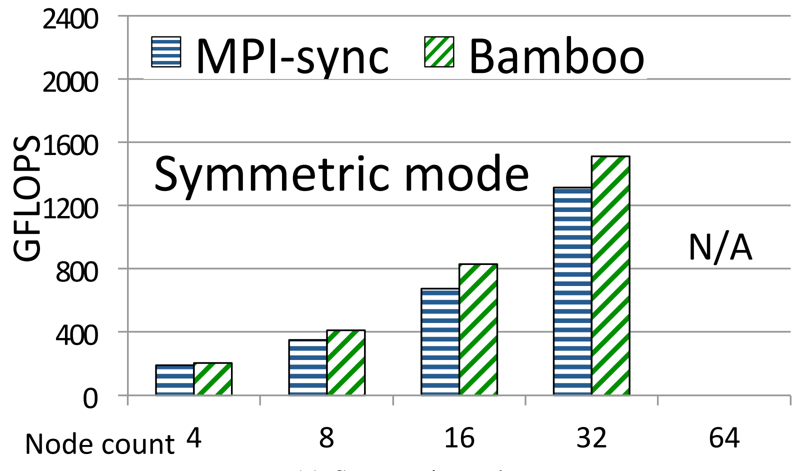
The benefit of Bamboo is more significant in MIC-MIC and symmetric modes, where the relative overhead of communication is large. By overlapping communication



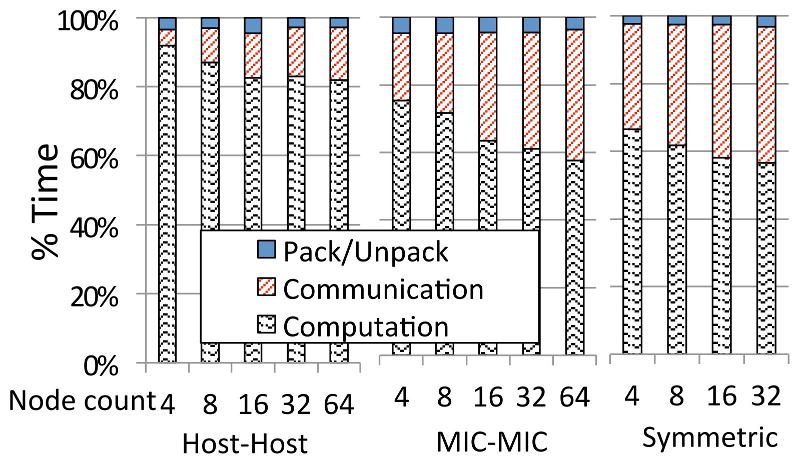
(a) Host-Host mode



(b) MIC-MIC mode

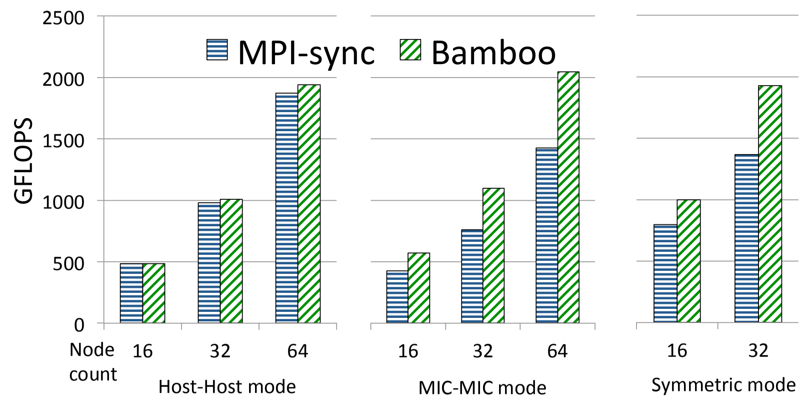


(c) Symmetric mode

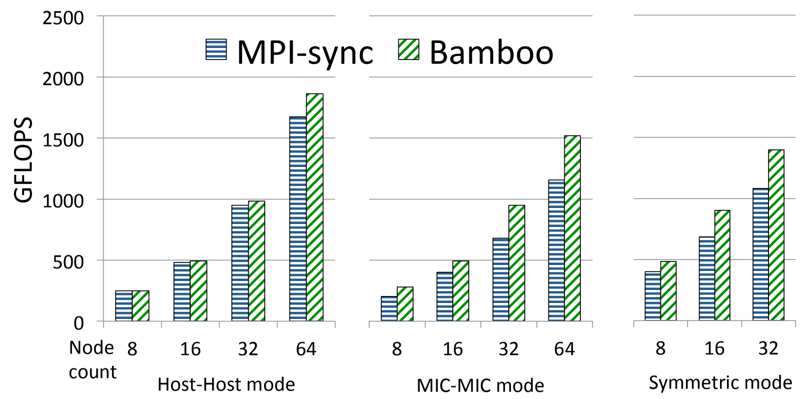


(d) Time distribution

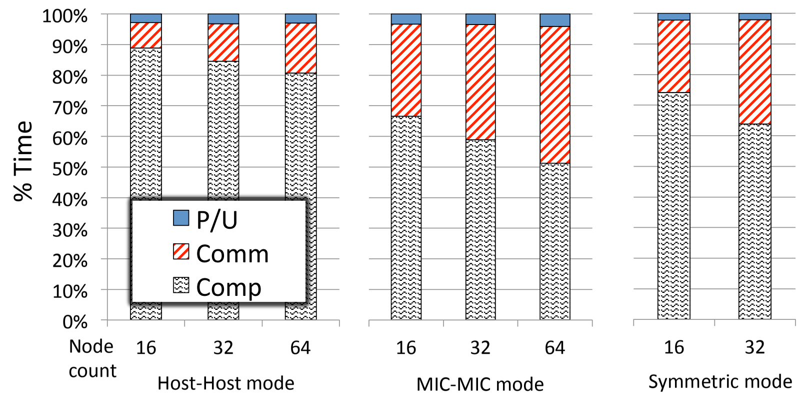
Figure 8.7. Weak scaling study with base problem size = 384x384x384. We increase the size equally on 3 dimensions, i.e. the problem size on P nodes is $\sqrt[3]{(P)} * 384 \times \sqrt[3]{(P)} * 384 \times \sqrt[3]{(P)} * 384$. We perform a 3D decomposition in this study.



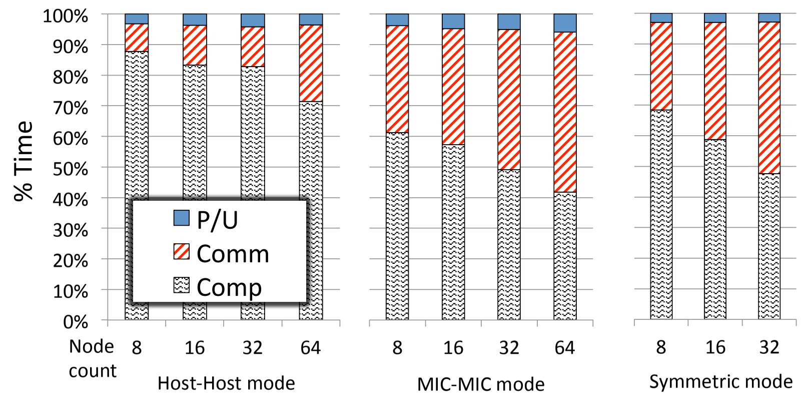
(a) Strong scaling performance. Probem size = 1536x1536x1536



(b) Strong scaling performance. Probem size = 1024x1024x1024



(c) Time distribution. Probem size = 1536x1536x1536



(d) Time distribution. Probem size = 1024x1024x1024

Figure 8.8. Strong scaling study: we test with two cubical problem sizes: 1536^3 and 1024^3. Since the leading dimension is large, a 3D decomposition is required

with computation, Bamboo improves the scalability of the MIC-MIC mode to the level of Host-Host mode. On 64 nodes, Bamboo in both modes runs with 2 Tflops, yielding a 87% efficiency (relative to the performance on 1 node). The improvement of Bamboo in the MIC-MIC mode is 33% on 16 MICs, 39% on 32 MICs, and 43% on 64 MICs. With the symmetric mode, Bamboo also performs well. The communication overhead in this mode is smaller in the MIC-MIC mode, so the relative improvement of Bamboo is also smaller: 24% on 16 MICs and 20% on 32 MICs. However, recall that symmetric mode runs roughly 2 times faster than MIC-MIC, making the absolute performance improvement of Bamboo in this mode comparable to it is in the MIC-MIC mode.

Strong scaling study

While we used weak scaling to adapt to large problem sizes and maintain good performance, strong scaling serves as a stress test for the performance of an application when communication cost grows. To this end, we used 2 problem sizes 1536^3 and 1024^3 to conduct strong scaling studies. Fig. 8.8 presents results of strong scaling studies. We observed that the performance with 1024^3 was lower than with 1536^3 in most configurations. This is reasonable since a large problem size has a lower surface over volume ratio than a smaller one, reducing the relative overhead of communication.

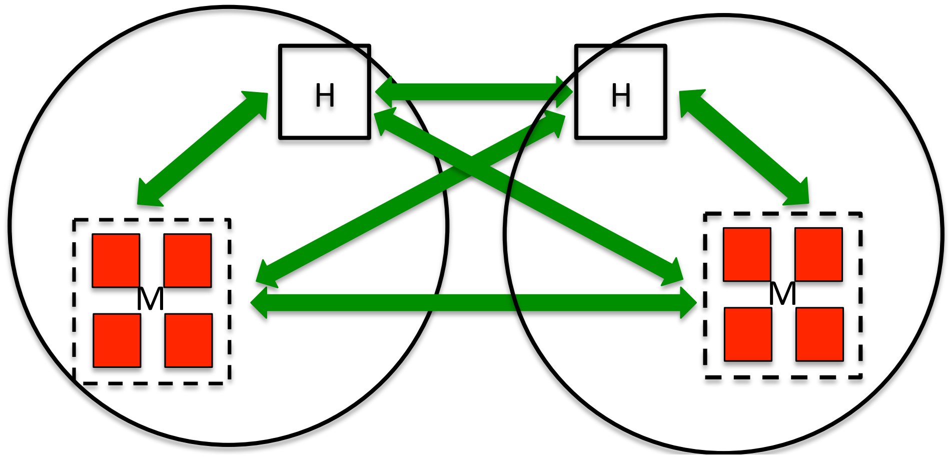
Compared to weak scaling study, Bamboo plays a more significant role in keeping the application scalable, especially with the MIC-MIC and symmetric modes. For MIC-MIC mode, Bamboo improves the performance by 44% with the large and 48% with the small problem sizes (both correspond to the 32-node configuration). For symmetric mode, *MPI_sync* performs well with the large problem size. Bamboo does even better when making this mode almost twice as fast as the MIC-MIC mode. Bamboo respectively improves the performance of *MPI_sync* by 25% and 41% on 16 and 32 nodes. Recall that symmetric is much faster than MIC-MIC and Host-Host modes, making the improvement of Bamboo in symmetric mode much more significant if we convert to absolute values. With the small problem size, *MPI_sync* performs well on 8 nodes, operating 1.6X and 2X faster than Host-Host and MIC-MIC modes, respectively. However, this variant slows down on 32 nodes and does not realize good performance on 64 nodes. This problem is due to a low device-to-host bandwidth that we will show in the next section. Bamboo improves *MPI_sync* by 32% on 16 nodes and 29% on 32 nodes. These results with small and large problem sizes demonstrate that Bamboo can hide communication overhead in both scalable and non-scalable situations. For Host-Host mode, similar to weak scaling we observed the benefit of Bamboo when communication overhead is significant. For example, we observed the improvement with Bamboo on 64-node configuration with both problem sizes.

8.3.4 Load balancing

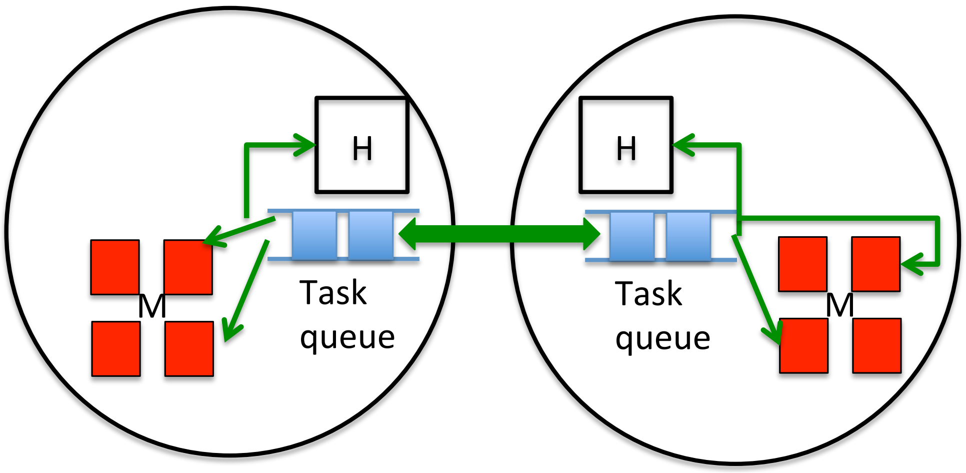
Although host and coprocessor coincidentally operate *3D Jacobi* at comparable rates on Stampede, this fact can change in other applications and on different systems, such as Tianhe-2, which has 3 Intel Xeon Phi and 2 Ivy Bridge processors per node [114]. In this section, we propose a new execution model to rectify the host-coprocessor discrepancy so that we can continue using regular distribution effectively on both host and coprocessor altogether.

Fig. 8.9(a) depicts the use of symmetric mode with 2 nodes, where hosts (H) and MICs (M) serve as SMP nodes. The effectiveness of this configuration relies on the assumption that host and coprocessor deliver similar performance. However, for reasons stated above, this assumption may not hold.

Since the imbalance occurs within a single node only, we can avoid the costs of migrating tasks, which is a more challenging solution to implement. Fig. 8.9(b) presents a modified version of symmetric mode. This scheme employs only one MPI process



(a) Symmetric mode



(b) A proposed scheme to rectify symmetric mode

Figure 8.9. Automatic load balancing in symmetric mode. Virtualization plays a significant role in balancing regular workload on heterogeneous processors

running on the host to communicate across nodes. Load balancing within node relies on the virtualization provided by Bamboo and on the dynamic scheduling supported by its runtime. In particular, Bamboo virtualizes MPI processes into many smaller homogeneous tasks. The runtime system employs a single queue per node to handle these tasks. This queue can be configured as a first-come-first-serve or a priority queue. Coprocessor and host serve as workers and keep picking tasks until there is no available task in the queue.

Implementing rectified symmetric mode requires that the host and co-processor residing on the same node see a single unified address space. Implementing such a single address space, either on hardware or by a software solution, requires significant efforts. We next present our first step in demonstrating the proposed scheme via a simulation using coprocessors only, which enables us to avoid the thorny implementation issues in supporting the single address space, while demonstrating the utility of the approach.

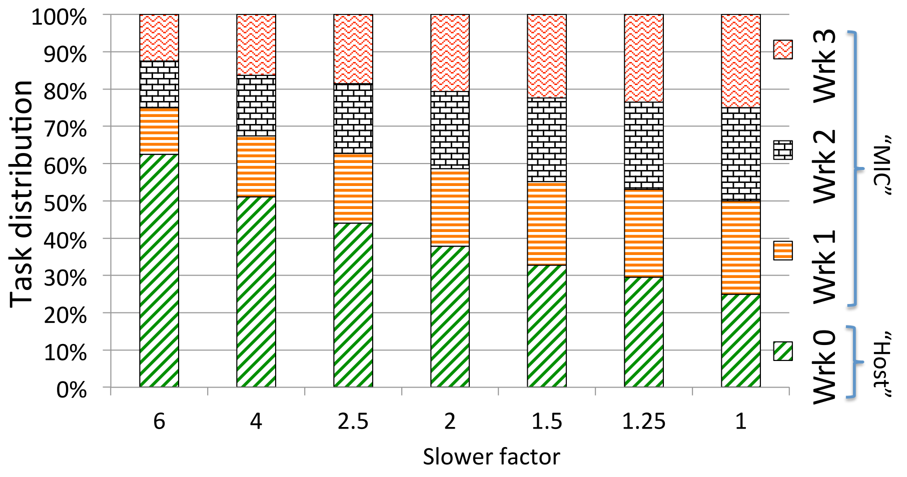
To simulate the unified address space between host and coprocessor, we extracted 1/4 number of cores on each coprocessor to use as host. As a result, each simulated coprocessor consists of 3-time cores more than its simulated host, a reasonable fraction to distinguish multi- and many-core processors. To simulate the fact that each core of the simulated host is faster than each core of the coprocessor, we dynamically added dummy computations to tasks at the time they are scheduled on the simulated coprocessors. We configured Tarragon with 4 worker threads, each spawning 60 OpenMP threads. We used the notion *slowdown factor* σ to denote how slower a worker thread on coprocessor is compared to that on the host. For $\sigma = 1$, all worker threads ran with the same rate, meaning that coprocessor is 3-time faster than host. For $\sigma = 6$, the coprocessor ran at only 3*(1/6)= 1/2 of the host's rate.

Fig. 8.10(a) presents the task distribution on worker threads of simulated hosts and coprocessors when the σ varies. We selected values for σ so that the coprocessor could be slower or faster than its host. We can see that when $\sigma = 6$, the host's worker thread is very fast compared to those of the coprocessor. In such a scenario, the scheduler dynamically assigns more tasks to the host's worker thread, thereby balancing the workload to maintain good performance. It is important to note that the task scheduling is purely driven by workload and the slowdown factor, and there is no intervention from the programmer. We also observed that the task distribution results shown in Fig. 8.10(a) hold for both single and multi-node configuration.

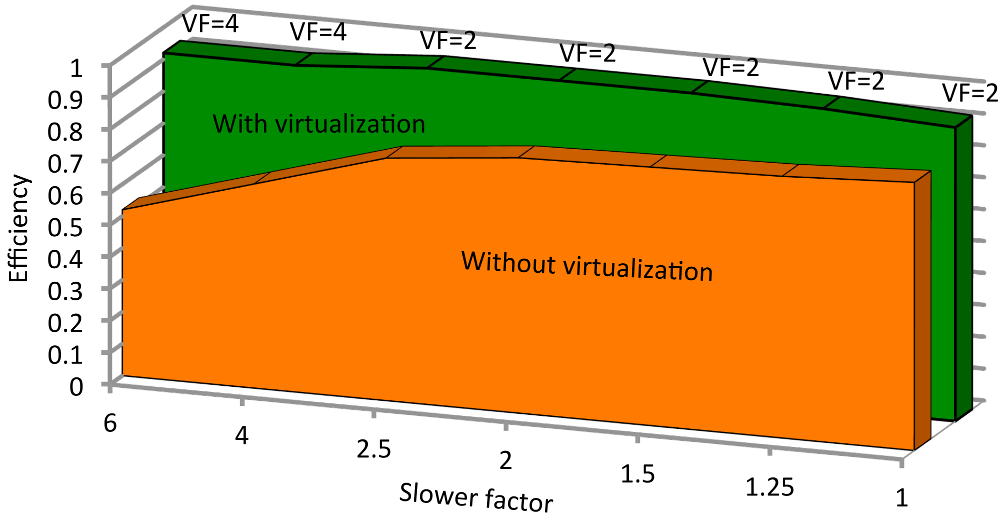
Fig. 8.10(b) demonstrates that latency hiding does not negatively impact load balance, but rather works in synergy with load balancing activity to improve performance. In particular, when $\sigma = 1$, entire performance benefit is due to latency hiding. With higher values of σ the load-balancing scheme contributes additional benefit, increasing the amount of performance improvement. In this study, we also observed the vital role of virtualization in both hiding latency and balancing the workload. The degree of virtualization is denoted by *virtualization factor* (VF), the total number of tasks divided by the total number of worker threads. Fig. 8.10(b) shows that a virtualization factor of 2 is sufficient when the discrepancy between worker threads is not significant. However, when the host's worker thread becomes much faster, a virtualization factor of 4 is required. This makes sense since we need more tasks to fill the larger performance gap among worker threads.

8.4 Summary

1. Advanced node architectures allow the programmer to accelerate computation kernels. However, this performance amplification in turn increases significantly communication costs relative to computation. As a result, hiding communication becomes more and more important than it was before. However, the complex memory hierarchy employed by advanced node architectures challenges any attempt to



(a) Virtualization help balance the workload across non-uniform processors



(b) Efficiency (performance/sustainablePerformance) with and without virtualization on 64 simulated nodes

Figure 8.10. The role of virtualization in enabling homogeneous programming in a heterogeneous computing environment

hide communication costs.

2. Graphics Processing Unit (GPU) is a device architecture commonly used to accelerate not only computer graphics but also many scientific computing applications. MPI+CUDA is a dominating programming model to parallelize applications on multiple GPUs distributed across the network. However, the traditional MPI+CUDA programming model requires the programmer to explicitly transfer data between host and GPU in order to communicate among GPUs. We proposed a new model that enables a peer-to-peer communication scheme between GPUs. Integrating the new model with Bamboo and Tarragon extensions, we are able to hide both host-host and host-device communication delays with computation.

3. Many Integrated Core (MIC) is a new processor architecture developed by Intel. Intel Xeon Phi is the first product of MIC and is referred to as coprocessor. On compute nodes consisting of both processors (e.g. Sandy Bridge processors) and coprocessors, the programmer can employ many execution modes. Bamboo currently supports 3 modes: host-host, MIC-MIC, and symmetric. On these 3 modes, we have demonstrated that Bamboo can significantly improve the performance by overlapping communication with computation. Since host and coprocessor can run at different speeds, we propose a new execution model to rectify the host-coprocessor discrepancy occurring in the symmetric mode. This scheme employs the host to communicate across nodes. Load balancing within node relies on the virtualization provided by Bamboo and on the dynamic scheduling supported by its runtime. Fig. 8.6 shows the performance in GFLOP/s of all code variants.

8.5 Acknowledgements

This chapter, in part, uses the material appeared in the Third International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC'13) with the title ”Bamboo - Preliminary scaling results on multiple hybrid nodes of Knights Corner and Sandy Bridge processors” by Tan Nguyen and Scott Baden.

This chapter, in part, is currently being prepared for submission for publication with Scott Baden. The dissertation author is the primary investigator and author of this material.

Chapter 9

Conclusion and future work

9.1 Research contributions

This dissertation presented a novel interpretation of Message Passing Interface to execute MPI applications under a data-driven model that can overlap communication with computation automatically. This interpretation factors scheduling issues and communication decisions out of program execution. Specifically, by reformulating MPI source into the form of a task dependency graph, which maintains the data dependency among tasks of the graph, we can rely on a runtime system to schedule tasks based on the availability of data and computing resources.

To implement our approach we developed Bamboo, a custom source-to-source translator that transforms MPI code into the task dependency graph representation. Bamboo treats the MPI API as an embedded domain specific language, and it requires only a modest amount of programmer annotation. Bamboo comprises 2 software layers: *core message passing* and *utility* layers. The *core message passing* layer transforms a minimal subset of MPI point-to-point primitives, whereas the *utility* layer implements high-level routines by breaking them into their point-to-point components, which will be then translated by the *core message passing* layer. Such a multi-layer design allows one to customize the implementation of MPI high-level routines. In addition, this design can reduce the amount of programming effort needed to port the *core message passing* layer to a different runtime system.

We demonstrated that Bamboo improved performance on three important application motifs: dense linear algebra, structured and unstructured grids. For dense linear algebra, we translated two well-known algorithms for computing dense matrix multiplication (Cannon and SUMMA algorithms). We also translated High Performance Linpack, a well-known benchmark that solves systems of dense linear equations using LU factorization. For structured grid, we translated an iterative solver for Poisson's equation and a geometric multigrid solver for Helmholtz's equation. Finally, for unstructured grid, we translated a hydrodynamics code to solve the Sedov blast wave problem. For all applications, we have validated our claim that, by interpreting an MPI program in terms of data flow execution, we can overlap communication with computation and thereby improving the performance significantly. Moreover, Bamboo performance meets or exceeds that of labor-intensive hand coding, at scale. Bamboo also improves performance of *communication avoiding* matrix multiplication (2.5D Cannon's algorithm). The result on this application demonstrates that the translated code not only avoids communication, but tolerates what it cannot avoid. We believe that this dual strategy will become more widespread as data motion costs continue to grow.

We also validated Bamboo on advanced node architectures, which accelerate node performance by offloading compute-intensive kernels to devices such as NVIDIA's GPU and Intel's MIC. On GPU clusters, Bamboo not only improves performance of a program written under the MPI+CUDA programming model, but also offers a simpler interface that allows communication between GPUs to be transparent to the programmer. On MIC-based clusters, Bamboo currently supports 3 execution modes. Experimental results demonstrate that Bamboo can improve the performance of MPI significantly with only a modest amount of programmer annotation.

Lastly, but certainly not least, Bamboo allows the programmer to specify scheduling hints as task priorities in order to optimize the scheduler. A task with higher priority will have a higher chance to be scheduled quickly. Such task prioritization support is important in applications that consist of irregular workloads. While Bamboo's scheduler employs a non-preemptive task scheduling [52–54], Bamboo allows tasks to voluntarily yield the processor at the time of its choosing, enabling tasks of the graph to work in a more cooperative manner. This dual scheduling scheme allows hardware resources to be efficiently shared among tasks. We evaluated the task prioritization support using two applications of the dense linear algebra motif: SUMMA matrix multiply and LU factorization. Experimental results demonstrated that we gained significant performance benefits by employing simple prioritization schemes.

9.2 Limitations

9.2.1 Unsupported MPI routines

Although the current implementation of Bamboo supports high-level routines such as collectives and communicator splitting, it does not yet support MPI derived datatypes (e.g. vector and indexed types) and virtual topologies (i.e. Cartesian grid and Graph). Although it is not mandatory to use these routines for developing an MPI program, such supports can make the programming task easier. Virtual topologies define the shape of MPI processes. Thus, the programmer can have an intuitive idea about how MPI processes are organized. In addition, the programmer can use many high-level communication operators which are pre-built for a particular topology. For example, the programmer can shift data along left-to-right edges of a one-dimensional process grid in one operation. Derived datatypes provide a means of describing the shape of sent and received data. Thus, derived datatypes reduce the need to marshal and demarshal data when communicating on scattered data sets.

9.2.2 Load balancing

Currently, Bamboo supports load balancing by virtualizing MPI processes into tasks and leveraging the dynamic scheduling support of the Tarragon runtime system. In Chapter 8, we have presented a simulation that shows a method to balance the workload assigned to both Sandy Bridge and MIC, which generally operate at different rates, by using dynamic scheduling. However, Bamboo hasn't supported this capability on a real platform. There are a few reasons for this limitation. First, host and accelerator/coprocessor often employ different ISAs. Second, the runtime system does not provide the *work stealing* support, an important technique for load balancing.

The application itself is another source of load unbalance. Indeed, for applications such as adaptive mesh refinement (AMR) and unbalanced tree search (UTS), computations are expected to be significantly unbalanced among processes. As a result, inter-process task migration support is crucial to balance the workload and realize expected performance. Bamboo and Tarragon, however, do not yet support task migration across memory address spaces. Tasks can migrate across worker threads residing in a common process only. Bamboo can be extended to support more irregular applications. This requires a stronger support from the runtime system, since task migration can be very costly due to data attached with the migrated computation. Thus, future extensions on Bamboo and its runtime system must be well designed to efficiently support task migration.

9.3 Future work

We can extend Bamboo to support complicated, heterogeneous computer systems. A future compute node may contain multiple types of multicore, or manycore processor, or both. Thus, processor cores may run at different speeds. Under this heterogeneous environment, data partitioning and mapping are non-trivial to the programmer. Bamboo alleviates these challenges by supporting process virtualization. However, in the future Bamboo needs to provide auto-tuning support for finding optimal or near-optimal process virtualization and task mapping. For irregular applications, hints from the programmer may be useful to effective task migration.

Appendix A

Testbed

A.1 Edison

Edison is a Cray XC30 system located at the National Energy Research Scientific Computing Center (NERSC). Edison contains 133,824 cores, organized into compute nodes each consisting of two 12-core Intel Ivy Bridge processors and 64GB of memory. Compute nodes are interconnected via Cray Aries interconnect with Dragonfly topology. All source code was compiled using the *CC* wrapper, with the *O3* optimization option. This wrapper is a front-end to MPI and we set it up to use the Intel compiler suite. High performance matrix multiply (*dgemm*) was supplied by the MKL library.

A.2 Hopper

Hopper is a Cray XE6 system, and it is also located at NERSC. Hopper consists of 153,216 cores packaged as dual socket 12-core AMD Magny-Cours 2.1GHz processors, which are further organized into two hex-core NUMA nodes. The 24-core compute nodes are interconnected via Gemini interconnect (a 3D toroidal topology). Each compute node contains 32GB of memory. Like on Edison, all source code was compiled using the CC wrapper, with the following optimization options: -O3 -ffast- math. Unless we describe explicitly, the reader can assume that we used the GNU compiler suite (GCC 4.6.1) on

Hopper. High performance matrix multiply (dgemm) was supplied by ACML version 4.4.0.

A.3 Stampede

Stampede is a hybrid CPU/MIC system located at the Texas Advanced Computing Center (TACC). This cluster consists of 6400 compute nodes, each equipped with two 8-core Sandy Bridge processors and one Intel Xeon Phi XE10P coprocessor (MIC). Each compute node of Stampede contains 32GB of host memory (NUMA). Nodes communicate via a Mellanox FDR InfiniBand interconnect with a 2-level fat-tree technology. We compile code with the Intel C++ compiler version 13.0. We use Mvapich to communicate among Sandy Bridge processors. To communicate among Sandy Bridge and MICs or between MICs, we use Intel MPI.

Stampede also consists of a few hybrid CPU/GPU system queues, each employing up to 32 GPU nodes. Each GPU node is equipped with a K20 GPU (Kepler) and two 8-core Intel Xeon E5 (Sandy Bridge) processors. Each GPU node also contains 32GB host memory and 5GB device memory. Nodes communicate via a Mellanox FDR InfiniBand interconnect. We use the Intel compiler to compile code running on host and CUDA 5.5 to compile GPU kernel code. We use Mvapich to communicate among GPU nodes.

Appendix B

Bamboo manual

B.1 Install ROSE and Bamboo

Bamboo is built on top of the ROSE compiler framework. Complete instructions to install ROSE and its dependencies can be found on ROSE's website [64]. Here we include a small set of instructions required to install Bamboo, assuming that the dependencies such as Boost, Java, and libtool are already available.

```
#Rose installation root
setenv ROSE_INSTALL path_to_rose

#Locate boost
setenv BOOST_ROOT path_to_boost

#Locate Java
setenv JAVA_HOME /opt/java/jdk1.7.0_03/

#Set load path
setenv LD_LIBRARY_PATH $BOOST_ROOT/lib:$LD_LIBRARY_PATH
setenv LD_LIBRARY_PATH $JAVA_HOME/jre/lib/amd64/server/:$LD_LIBRARY_PATH
setenv PATH $JAVA_HOME/bin:$PATH

#2. Untar rose, cd in, and run ./build
# Generates a new subdirectory containing the rose source directory
cd ..
mkdir buildrose
cd buildrose
```

```

#3. configure rose into your userspace from the newly created buildrose dir
../rose-0.9.5a-20584/./configure --prefix=$ROSE_INSTALL --with-gcc-omp --with-
    gomp_omp_runtime_library=/usr/lib --with-boost=$BOOST_ROOT

#4. Compile Rose (on 64-bit machines, be sure to modify -m32 to -m64)
make -j number_of_threads

#5. (optional) Check Rose
  make check
  make installcheck

#6. Install Rose
 make install

#7 After building Rose, finalize paths
setenv PATH $PATH:$ROSE_INSTALL/bin
setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:$ROSE_INSTALL/lib

#8. Build Bamboo
tar -xvf bamboo.tar
# In file configure.in, update paths to ROSE, BOOST, TARRAGON, and BAMBOO
make
make install

#Set path to Bamboo executable
setenv PATH $PATH:$BAMBOO_HOME/bin

#Translate an MPI input source
$BAMBOO_HOME/bin/bambooCC path_to_input_source
```

B.2 Annotate an MPI program

B.2.1 Bamboo canonical form

When a Bamboo program has been appropriately annotated, it has a syntactical structure called *Bamboo Canonical Form*. The annotations do not change the meaning of the program (other than, for example, to reorder floating point operations, which could produce different results due to roundoff) but they enable Bamboo to interpret the order of computatation and data motion differently from conventional MPI programs, in order to improve performance.

We construct a Context Free Grammar for the canonical form based on point-to-point communication primitives. Recall that we transform non-point-to-point communication routines, e.g. collective and communicator splitting, into their component sends and receives. The specific implementation of these communication algorithms is not mentioned, and is assumed to be correct.

```
P → COC
C → CC | O | c | ε
O → (E{C}E) | (E)
E → EE | [S] | <R> | ε
S → SS | s | w
R → RR | r | r S | w
```

The grammar can be interpreted as follows. A Bamboo conforming program P consists of one or more *olap-region O* interspersed with executable statements *c* that are free from MPI communication calls. Each *O* region may contain communication blocks and a single computational block *C* that may appear at any place in the *O* region. This computational block may contain other *O* regions. A communication block can be a send block *S* or a receive block *R*. *S* and *R* are blocks that hold MPI primitives. *S* contains Sends or iSends called *s* and Wait or Waitall called *w*. *R* contains Recv or iRecv called *r*, *w*, and one or more *r* followed by one or more *s*. Note that *s* and *r* may associate with affiliated code that sets up arguments to sends and receives, such as message destination and source, but in the case of receives, may not be long running, in the sense of an active message handler.

The Context Free Grammar only defines the syntax of *Bamboo Canonical Form*.

To make sure that the program works in a proper way, we add the following semantic note to this form. The totality of *R* blocks is independent of the totality of *S* Blocks. However, the order of instructions within a communication block and between communication blocks of the same type is preserved.

B.2.2 Bamboo directives

In the *Bamboo Canonical Form*, we used terminals (), {}, [], <> to mark *olap-regions*, communication and computation *blocks*. In practice, the Bamboo programmer annotates *olap-regions* and *blocks* with directives. All Bamboo directives begin with *#pragma bamboo*, followed by a directive name and any clauses associated with the directive. Bamboo provides two kinds of directives. The first kind treats *olap-regions* and computation/communication blocks as described previously. The syntaxes for the *olap-region* and the computation/communication blocks directives are as follows, where we use regular expression syntax to specify alternatives.

```
#pragma bamboo olap
{...}
#pragma bamboo [send|receive]
{...}
#pragma bamboo compute
{...}
```

Bamboo generates *firing* and *yielding rules* for an *olap-region* by extracting the following information from all *receive blocks* residing in the region: i) *source* and *tag* arguments of *MPI_Recv* and *MPI_Irecv* calls ii) associated statements that determine the value of *source* and *tag*. The code snippet below shows a *receive block*, in which *MPI_Recv* calls are governed by *for* and *if* statements. The programmer can either specify the expressions for calculating *source* and *tag* as arguments in MPI calls (line #4), or place the statements to compute these arguments in a basic block annotated by the *includeStatementBlockInTheFiringRule* pragma (lines #6 to #10). The programmer can place this pragma in the same or parent scopes of the receive routine.

```
#pragma bamboo receive
{
  for(int i=0; i<maxNeighbors; i++){
    if(isNeighbor(i)) MPI_Recv(..., foo_source(i), ..., foo_tag(i), ...);
    if(isNeighbor(i)){
      #pragma bamboo includeStatementBlockInTheFiringRule
      {
        source = foo_source(i);
        tag = foo_tag(i);
      }
      MPI_Recv(..., source, ..., tag, ...);
    }
  }
}
```

Bamboo allows receive calls to reside in a procedure. If this procedure takes arguments that will be used to determine *source* and *tag*, the pragma *includeStatementsAboveInTheFiringRule* must be placed at the begining of the procedure. This pragma includes all statements above it in the same scope, including arguments passed to a procedure.

```
void receive(int* neighbor, int* tag);
{
  #pragma bamboo includeStatementsAboveInTheFiringRule
  for(int i=0; i<maxNeighbors; i++){
    if(neighbor[i]) MPI_Recv(..., neighbor[i], ..., tag[i], ...);
}
```

Directives of the second kind, called *optimizations*, specify optimizations that may not be needed in all programs, though highly scalable programs will generally require them.

Communication layout clauses

Before a task dependency graph can run, the runtime system needs to know *graph topology*, which defines how tasks communicate with each other during their life time. This information will be maintained until the graph finishes its execution. By default, Bamboo lets each task memorize the information of all other tasks of the graph. However, the actual communication of a practical application can be very sparse. At scale, the graph construction time and the memory space needed for *graph topology* can explode, since we expect to have millions of tasks. Bamboo introduces the optional communication layout, where the programmer can provide a hint on the communication pattern used in each *olap-region*. This information will be used to reduce the time for graph construction and the space to store *graph topology* only; no inference from this information is made for optimizing the graph execution time.

The code snippet below shows the communication layout for a *nearest neighbor* communication pattern on a 3-dimensional process grid. With this information, at the graph construction time each task only contacts and memorizes information of 6 nearest neighbors, 2 for each dimension. Table B.1 shows communication layouts commonly used in practice. By default, the *AllToAllConnector* layout is used with a 1-dimensional process grid. This layout works for every application since it creates a full mesh of tasks.

```
#pragma bamboo dimension 3
#pragma bamboo olap layout OneNeighborConnector 0 1 2
{
  ...
}
```

Table B.1. Communication layouts commonly used in practice. The programmer can find or define more communication layouts in Tarragon at the following path *Tarragon_home/include/visitors/connectors.h*

Layout name	Arguments	Description
AllToAllConnector	Applying dimensions	Connect every task with every other task in the specified dimension
NearestNeighborConnector	N/A	Connect each task with its nearest neighbors in all dimensions
OneNeighborConnector	Applying dimensions	Connect each task with its nearest neighbors in the specified dimensions
RubricConnector	N/A	Applicable to 3D space only. Each task connects with all 26 surrounding neighbors
BinomialTreeConnector	N/A	All tasks form a binomial tree
MinHeapConnector	Applying dimensions	All tasks in each specified dimension form a binary tree

Task prioritization

The programmer can assign different priorities to different tasks, or to the same task but at different places of the program. The syntax to assign priority to a task is as follows. The programmer can insert priority pragma in any procedure that directly or indirectly contains *olap-region*.

```
#pragma bamboo priority value [priority Value] applyIf [if-specifier]
```

B.2.3 Library of collectives

The programmer has the freedom to customize the Bamboo implementation of collectives. The default implementation of common collectives is stored in the 'collectiveLib' directory. The programmer can add their implementation of missing routines or modify the implementation of current ones.

B.3 Configure the runtime system

The Tarragon runtime system can be configured in a few different ways [52]. This section reviews notable options in configuring the runtime system.

B.3.1 Single-threaded and multi-threaded modes

Tarragon can be configured to run with either *single-threaded* or *multi-threaded* mode. Let *m* be the number of available processor cores within a multi-core node. Under the *single-threaded* mode, each MPI process is responsible for handling the communication and executing tasks. Thus, to configure the runtime system with the *single-threaded* mode, we employ *m* MPI processes per node, one running on a processor core. Under the *multi-threaded* mode, a processor core is dedicated to handle communication while the remaining m-1 cores execute tasks, assuming that m $>$ 1. Thus, to configure the runtime system with the *multi-threaded* mode, we employ only 1 MPI process per node and m threads per process.

Tarragon sets the *multi-threaded* mode as the default mode. To switch to the *single-threaded* mode, the user must enable the macro *TGN_SINGLE* in the file 'include/tgn_config.h'.

B.3.2 Thread affinity

When using the *multi-threaded* mode, the Tarragon user can customize the binding of the communication handler thread and the worker threads. The binding of the communication handler thread is set by the *Worker::base* variable in the file 'system/tarragon.C'. For example, 'Worker:base = 0' binds the communication handler thread to processor core #0 of each compute node. The worker thread affinity is controlled by the *_affinity* variable in the file 'system/worker.C'. For example, '_affinity = Worker:base + Worker:_id + 1' binds worker threads (#0 to #W-1) to processor cores #1 to #W, dedicating the processor core #0 to the communication handler.

B.3.3 Message buffer size

The communication handler of the Tarragon runtime system buffers messages. In particular, it employs fixed size buffers to temporarily store incoming data before injecting them to tasks. Thus, each buffer has to be able to hold the largest message used in the application. The programmer can configure the message buffer size by modifying the *TGN_BUF_SIZE* macro in the file 'include/tgn_config.h'.

B.4 Compile and link the task graph program

We now can compile and link the generated task graph code against the Tarragon library. Below is a sample Makefile, in which mainfile.C is the code generated by Bamboo and otherfiles.C represents for other auxiliary files.

```
CXX = mpicxx
FLAGS += -O3 #and other flag options
TARRAGON_HOME = path_to_tarragon
TARRAGON_INC = -D__MPI -I$(TARRAGON_HOME)/include
TARRAGON_LIB = -L$(TARRAGON_HOME)/lib -ltgn

all:     exec
exec:   mainfile.o otherfiles.o
        $(CXX) $(FLAGS) mainfile.o otherfiles.o $(TARRAGON_LIB) -o exec

mainfile.o:   mainfile.C
        $(CXX) $(FLAGS) $(TARRAGON_INC) -c mainfile.C -o mainfile.o

otherfiles.o: otherfiles.C
        $(CXX) $(FLAGS) -c otherfiles.C -o otherfiles.o

clean:
        rm exec
        rm *.o
```

B.5 Launch the executable

The task graph program generated by Bamboo remains an MPI program. Thus, the programmer uses *mpirun* or another equivalent command such as *aprun* and *ibrun* to launch it. However, since the Bamboo translator reformulates the execution behavior of the original program, the arguments to launch the task graph program are slightly different from those of the original one.

Here we show the commands to run the task graph program with the multi-threaded and single-threaded modes. In the multi-threaded mode (command #1), we employ W worker threads per MPI process. Thus, the number of MPI processes required in this case is only P/W. In the single-threaded mode (command #2), we launch P processes, one on each processor core. In both cases, the programmer can specify the number of tasks by setting P_i. In order to hide latency and balance the workload among worker threads, P_i must be chosen such that $\Pi_0^{n-1}\ P_i$ is multiple of P.

```
mpirun -np P/W ./exec [progArgs] -tarragonPi Pi -w W #i=0..n-1
mpirun -np P ./exec [progArgs] -tarragonPi Pi #i=0..n-1
```

Where,

- P is the number of processor cores
- W is the number of worker threads per MPI process
- exec is the ELF file obtained by linking the compiled task graph program with the Tarragon library
- progArgs represents arguments associated with the original MPI program
- n is the number of dimensions of the process grid employed by the original MPI program
- $P_i, i = 0..n-1$ is the number of tasks on dimension i such that $\Pi_0^{n-1}\ P_i$ is multiple of P

Appendix C

Case study

C.1 2D Jacobi

In this chapter we use *2D Jacobi* as a case study for the application of Bamboo. The *2D Jacobi* code that we use is a PDE solver that iteratively sweeps a 5-point stencil operation over a 2D finite element mesh. Due to the data dependency of the stencil operation, MPI processes exchange boundary data with their neighbors in the Manhattan directions. Figure C.1 shows how an MPI process communicates with its neighbors. We next present the MPI code for 2D Jacobi annotated with Bamboo pragmas.

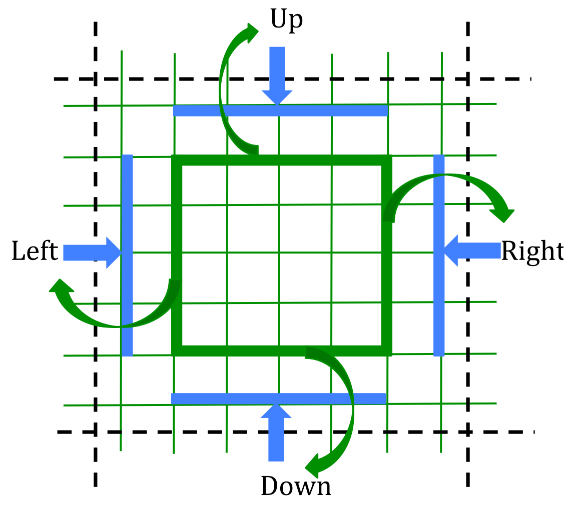


Figure C.1. 2D Jacobi solver using a 5-point stencil update scheme. Each MPI process sends and receives different regions of data

C.2 Annotate the 2D Jacobi code

```
Process Grid: P = Px * Py //input by the programmer
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int rankx= rank%px;
int ranky= rank/px;
enum Direction{left=0, right, up, down};
int neighbor[4];
neighbor[left] = rankx > 0 ? rank - 1:-1;
neighbor[right] = rankx<(Px-1) ? rank + 1:-1;
neighbor[up] = ranky<(Py-1)? rank + Px :-1;
neighbor[down] = ranky > 0 ? rank - Px :-1;
#pragma dimension 2
for (int iter=0; iter<nIters; iter++){
  #pragma bamboo olap layout NearestNeighborConnector
  {
    #pragma bamboo receive
    {
     Pack data
     if(neighbor[left]) MPI_Irecv(bufferLeft_recv, neighbor[left], leftTAG,...);
     if(neighbor[right]) MPI_Irecv(bufferRight_recv, neighbor[right], rightTAG,...);
     if(neighbor[up]) MPI_Irecv(bufferUp_recv, neighbor[up], upTAG,...);
     if(neighbor[down]) MPI_Irecv(bufferDown_recv, neighbor[down], downTAG,...);
    }
    #pragma bamboo send
    {
     if(neighbor[left]) MPI_Isend(bufferLeft_send, neighbor[left], leftTAG,...);
     if(neighbor[right]) MPI_Isend(bufferRight_send, neighbor[right], rightTAG,...);
     if(neighbor[up]) MPI_Isend(bufferUp_send, neighbor[up], upTAG,...);
     if(neighbor[down]) MPI_Isend(bufferDown_send, neighbor[down], downTAG,...);
     MPI_Waitall(request_count, request, MPI_STATUS_IGNORE);
    }
  }
  Upack data
  5-point stencil update
  swap (Uold, Unew);
}
local_residual = residual(Uold);
MPI_Reduce(&local_residual, &global_residual, ...);
```

In the MPI code above, MPI processes use point to point communication routines to exchange data with their neighbors at every time step. Thus, we annotated these calls with an *olap-region*. Since MPI processes send and receive different data, we placed send calls in a *send* block and receive calls in a *receive* block. MPI_Reduce is a collective call, so we didn't have to annotate it. The *NearestNeighborConnector* layout establishes a connection between each task with its 4 nearest neighbors, reducing the space needed to store the *graph topology*.

C.3 Launch the 2D Jacobi code variants

C.3.1 MPI

With pure MPI, we spawn P MPI processes running on P processor cores, organized in a $P_X \times P_Y$ process grid where P = P_X * P_Y (command #1). If the programmer employs OpenMP (or another multithreading library) to parallelize the in-node computation, the number of MPI processes will be reduced to the number of nodes. Specifically, with MPI+OpenMP we spawn N MPI processes running on N nodes, organized in a $N_X \times N_Y$ process grid where N = N_X * N_Y (command #2).

```
1  mpirun -np P ./Jac_MPI -px PX -py PY
2  mpirun -np N ./Jac_MPIOpenMP -px Nx -py Ny
```

C.3.2 Bamboo

Bamboo always employs 1 process per node (or per socket if each compute node contains multiple sockets). Each process spawns P/N worker threads, where P is the total number of available processor cores and N is the number of compute nodes. If the compute node does not have an extra processor core to host the runtime system, the programmer should spawn only P/N - 1 worker threads per process, dedicating one core to the runtime system. Command #1 shows command-line arguments that don't employ virtualization. With this command, there are only P tasks executed by N * (P/N) = P worker threads. In this scenario, if each task does not use any finer grain decomposition and pipeline algorithm (such as the SUMMA algorithm shown in Sec. 5.2.3), no communication overlap can be realized. The common case, therefore, is to use virtualization to realize communication overlap automatically. Command #2 shows how we can employ (kx*P_X) * (ky*P_Y) tasks that will be run by P worker threads, where kx*ky is larger than 2. Commands #3 and #4 show command-line arguments for Bamboo+OpenMP without and with virtualization, respectively. In these scenarios, each process employs only 1 worker thread.

```
1 mpirun -np N ./Jac_bamboo -px Px -py Py -tarragonP0 Px -tarragonP1 Py -w P/N
2 mpirun -np N ./Jac_bamboo -px kx*Px -py ky*Py -tarragonP0 kx*Px -tarragonP1 ky*Py -
      w P/N
3 mpirun -np N ./Jac_bambooOpenMP -px Nx -py Ny -tarragonP0 nx -tarragonP1 ny -w 1
4 mpirun -np N ./Jac_bambooOpenMP -px kx*Nx -py ky*Ny -tarragonP0 kx*Nx -tarragonP1
      ky*Ny -w 1
```

Appendix D

Side effects of CUDA routines

Tasks sharing the same *device* may call to some routines that have side effects on each other or on the runtime system. The side effects can lead to either correctness or performance issues. Table D.1 lists CUDA calls commonly used by the programmer that can cause such side effects. Except for *cudaDeviceReset*, other routines listed in the table may lead to performance reduction if they are invoked during the course of computation. The reason is that a call to these routines will block the whole *device*. As a result, data transfer operations between host and *device* issued by the communication handler and GPU kernel calls issued by worker threads have to wait until these calls complete.

Multiple calls to *cudaDeviceReset* cause segmentation fault since a call to this routine deallocates all resources on the device. Thus, we recommend that the programmer should not use this routine. To deallocate memory, *cudaFree* is a better solution since this routine revokes only the memory previously allocated to the caller task.

Table D.1. Side effects of translating code containing special CUDA calls

Routine type	Routine name	Side effect description
Device Management	cudaDeviceSynchronize	Block the device
	cudaDeviceSetCacheConfig	Block the device if cache configuration changes
	cudaDeviceSetSharedMemConfig	Block the device if SM configuration changes
	cudaDeviceReset	Deallocate resources and reset state on the device. The programmer should not use this routine since segmentation fault occurs when 2 or more tasks invoke this routine on the same device.
Thread Management	cudaThreadSetCacheConfig	Similar to cudaDeviceSetCacheConfig
	CudaThreadSynchronize	Similar to cudaDeviceSynchronize
Memory Management	cudaMalloc(3D/Array/Pitch)	Block the device
	Blocking cudaMemcpy variants	Block the device
	cudaFree	Block the device
	Blocking cudaMemset variants	Block the device

Bibliography

[1] Culler D, Karp R, Patterson D, Sahay A, Schauser KE, Santos E, et al. LogP: Towards a Realistic Model of Parallel Computation. In: Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPOPP '93. New York, NY, USA: ACM; 1993. p. 1–12. Available from: http://doi.acm.org/10.1145/155332.155333.

[2] van de Geijn RA, Watts J. SUMMA: Scalable Universal Matrix Multiplication Algorithm. Austin, TX, USA; 1995.

[3] Williams S, Kalamkar DD, Singh A, Deshpande AM, Van Straalen B, Smelyanskiy M, et al. Optimization of geometric multigrid for emerging multi- and manycore processors. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. SC '12; 2012. p. 96:1–96:11.

[4] Titus G. Exploring Co-Design in Chapel Using LULESH;. Available from: http://chapel.cray.com/presentations/Chapel-LULESH-SIAM-CSE-2013.pdf.

[5] Thakur R, Rabenseifner R. Optimization of Collective communication operations in MPICH. International Journal of High Performance Computing Applications. 2005;19:49–66.

[6] Top 500 supercomputer sites;. Available from: http://www.top500.org.

[7] InfiniBand Roadmap;. Available from: http://www.infinibandta.org/content/pages.php?pg=technology_overview.

[8] Shalf J, Dosanjh S, Morrison J. Exascale computing technology challenges. In: Proceedings of the 9th international conference on High performance computing for computational science. VECPAR'10. Berlin, Heidelberg: Springer-Verlag; 2011. p. 1–25.

[9] Bosschere KD, D'Hollander EH, Joubert GR, Padua D, Peters F. Applications, Tools and Techniques on the Road to Exascale Computing. Amsterdam, The

Netherlands, The Netherlands: IOS Press; 2012.

[10] Sarkar V, Harrod W, Snavely AE. Software challenges in extreme scale systems. In: Journal of Physics: Conference Series. vol. 180. IOP Publishing; 2009. p. 012045.

[11] Hoare CAR. Communicating sequential processes. Communications of the ACM. 1978;21(8):666–677.

[12] Valiant LG. A bridging model for parallel computation. Commun ACM. 1990 August;33:103–111. Available from: http://doi.acm.org/10.1145/79173.79181.

[13] Wen C, Yelick K. Portable Runtime Support for Asynchronous Simulation. In: in International Conference on Parallel Processing; 1995. .

[14] Krishnamurthy A, Culler DE, Dusseau A, Goldstein SC, Lumetta S, von Eicken T, et al. Parallel Programming in Split-C. In: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing. Supercomputing '93. New York, NY, USA: ACM; 1993. p. 262–273. Available from: http://doi.acm.org/10.1145/169627.169724.

[15] Chen WY, Iancu C, Yelick K. Communication Optimizations for Fine-Grained UPC Applications. In: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques. PACT '05. Washington, DC, USA: IEEE Computer Society; 2005. p. 267–278. Available from: http://dx.doi.org/10.1109/PACT.2005.13.

[16] Maruyama N, Nomura T, Sato K, Matsuoka S. Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-scale GPU-accelerated Supercomputers. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. SC '11. New York, NY, USA: ACM; 2011. p. 11:1–11:12. Available from: http://doi.acm.org/10.1145/2063384.2063398.

[17] Datta K, Murphy M, Volkov V, Williams S, Carter J, Oliker L, et al. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: Proceedings of the 2008 ACM/IEEE conference on Supercomputing. SC '08. Piscataway, NJ, USA: IEEE Press; 2008. p. 4:1–4:12. Available from: http://dl.acm.org/citation.cfm?id=1413370.1413375.

[18] Nguyen A, Satish N, Chhugani J, Kim C, Dubey P. 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Net-

working, Storage and Analysis. SC '10. Washington, DC, USA: IEEE Computer Society; 2010. p. 1–13. Available from: http://dx.doi.org/10.1109/SC.2010.2.

[19] Wittmann M, Hager G, Wellein G. Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory. In: Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on; 2010. p. 1–7.

[20] Wellein G, Hager G, Zeiser T, Wittmann M, Fehske H. Efficient Temporal Blocking for Stencil Computations by Multicore-Aware Wavefront Parallelization. In: Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International. vol. 1; 2009. p. 579–586.

[21] Sawdey AC, OKeefe MT, Jones WB. A General Programming Model for Developing Scalable Ocean Circulation Applications. In: IN PROCEEDINGS OF THE ECMWF WORKSHOP ON THE USE OF PARALLEL PROCESSORS IN METEOROLOGY; 1997. p. 209–225.

[22] Danalis A, Pollock L, Swany M. M.: Automatic MPI application transformation with ASPhALT. In: In: Par. and Distr. Proc. Symp., IPDPS 2007; 2007. .

[23] Kurzak J, Dongarra J. Fully Dynamic Scheduler for Numerical Computing on Multicore Processors LAPACK Working Note 220;.

[24] Bosilca G, Bouteiller A, Danalis A, Faverge M, Haidar A, Herault T, et al. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. In: Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on; 2011. p. 1432–1441.

[25] Marjanović V, Labarta J, Ayguadé E, Valero M. Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In: Proceedings of the 24th ACM International Conference on Supercomputing, ICS '10; 2010. p. 5–16.

[26] Chatterjee S, Tasirlar S, Budimlic Z, Cave V, Chabbi M, Grossman M, et al. Integrating Asynchronous Task Parallelism with MPI. In: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing. IPDPS '13; 2013. p. 712–725.

[27] Colella P. Defining Software Requirements for Scientific Computing; 2004.

[28] Karlin I, Keasler J, Neely R. LULESH 2.0 Updates and Changes; 2013. LLNL-

TR-641973.

[29] Bilardi G, Herley KT, Pietracaprina A, Pucci G, Spirakis P. BSP vs LogP. In: Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures. SPAA '96. New York, NY, USA: ACM; 1996. p. 25–32. Available from: http://doi.acm.org/10.1145/237502.237504.

[30] Goudreau M, Lang K, Rao S, Suel T, Tsantilas T. Towards Efficiency and Portability: Programming with the BSP Model. In: Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures. SPAA '96. New York, NY, USA: ACM; 1996. p. 1–12. Available from: http://doi.acm.org/10.1145/237502.237503.

[31] Goudreau M, Hill JMD, Lang K, Mccoll B, Rao SB, Stefanescu DC, et al.. A Proposal for the BSP Worldwide Standard Library (preliminary version);.

[32] Geist A, Beguelin A, Dongarra J, Jiang W, Manchek R, Sunderam V. PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing. Cambridge, MA, USA: MIT Press; 1994.

[33] Message Passing Interface Forum;. Available from: http://www.mpi-forum.org.

[34] El-Ghazawi T, Carlson W, Sterling T, Yelick K. UPC: Distributed Shared-Memory Programming. Wiley-Interscience; 2003.

[35] Numrich RW, Reid J. Co-array Fortran for Parallel Programming. SIGPLAN Fortran Forum. 1998 August;17(2):1–31. Available from: http://doi.acm.org/10.1145/289918.289920.

[36] Hilfinger PN, Bonachea D, Gay D, Graham S, Liblit B, Pike G, et al. Titanium Language Reference Manual. Berkeley, CA, USA; 2001.

[37] Allen E, Chase D, Hallett J, Luchangco V, Maessen JW, Ryu S, et al. The Fortress Language Specification. Sun Microsystems, Inc.; 2007. Available from: http://research.sun.com/projects/plrg/Publications/fortress1.0beta.pdf.

[38] Chamberlain BL, Callahan D, Zima HP. Parallel Programmability and the Chapel Language. Int J High Perform Comput Appl. 2007 August;21(3):291–312. Available from: http://dx.doi.org/10.1177/1094342007078442.

[39] Charles P, Grothoff C, Saraswat V, Donawa C, Kielstra A, Ebcioglu K, et al. X10: An Object-oriented Approach to Non-uniform Cluster Computing. SIGPLAN

Not. 2005 October;40(10):519–538. Available from: http://doi.acm.org/10.1145/1103845.1094852.

[40] Dennis JB, Misunas DP. A Preliminary Architecture for a Basic Data-flow Processor. In: 25 Years of the International Symposia on Computer Architecture (Selected Papers). ISCA '98. New York, NY, USA: ACM; 1998. p. 125–131. Available from: http://doi.acm.org/10.1145/285930.286058.

[41] Gurd JR, Kirkham CC, Watson I. The Manchester Prototype Dataflow Computer. Commun ACM. 1985 January;28(1):34–52. Available from: http://doi.acm.org/10.1145/2465.2468.

[42] Whiting PG, Pascoe RSV. A History of Data-Flow Languages. IEEE Ann Hist Comput. 1994 Dec;16(4):38–59. Available from: http://dx.doi.org/10.1109/85.329757.

[43] Blumofe RD, Joerg CF, Kuszmaul BC, Leiserson CE, Randall KH, Zhou Y. Cilk: An Efficient Multithreaded Runtime System. SIGPLAN Not. 1995 August;30(8):207–216. Available from: http://doi.acm.org/10.1145/209937.209958.

[44] Blumofe RD, Leiserson CE. Scheduling Multithreaded Computations by Work Stealing. J ACM. 1999 September;46(5):720–748. Available from: http://doi.acm.org/10.1145/324133.324234.

[45] Graham RL. Bounds on Multiprocessing Anomalies and Related Packing Algorithms. In: Proceedings of the May 16-18, 1972, Spring Joint Computer Conference. AFIPS '72 (Spring). New York, NY, USA: ACM; 1972. p. 205–217. Available from: http://doi.acm.org/10.1145/1478873.1478901.

[46] Kurzak J, Luszczek P, YarKhan A, Faverge M, Langou J, Bouwmeester H, et al. Multithreading in the PLASMA Library. 2013;.

[47] YarKhan A, Kurzak J, Dongarra J. Quark users guide: Queueing and runtime for kernels. University of Tennessee Innovative Computing Laboratory Technical Report ICL-UT-11-02. 2011;.

[48] Agha G. An overview of actor languages. SIGPLAN Not. 1986 June;21:58–67. Available from: http://doi.acm.org/10.1145/323648.323743.

[49] Hewitt C, Bishop P, Steiger R. A universal modular ACTOR formalism for artificial intelligence. In: Proceedings of the 3rd international joint conference on

Artificial intelligence. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 1973. p. 235–245. Available from: http://portal.acm.org/citation.cfm?id=1624775.1624804.

[50] Kale LV, Krishnan S. CHARM++: a portable concurrent object oriented system based on C++. In: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications. OOPSLA '93. New York, NY, USA: ACM; 1993. p. 91–108. Available from: http://doi.acm.org/10.1145/165854.165874.

[51] Huang C, Kale LV. Charisma: Orchestrating Migratable Parallel Objects. In: Proceedings of IEEE International Symposium on High Performance Distributed Computing (HPDC); 2007. .

[52] Cicotti P. Tarragon: a Programming Model for Latency-Hiding Scientific Computations. Department of Computer Science and Engineering, University of California, San Diego; 2011.

[53] Cicotti P, Baden SB. Asynchronous programming with Tarragon. In: Proc. 15th IEEE International Symposium on High Performance Distributed Computing. Paris, France; 2006. p. 375–376.

[54] Cicotti P, Baden SB. Latency Hiding and Performance Tuning with Graph-Based Execution. In: Data-Flow Execution Models for Extreme Scale Computing (DFM), 2011 First Workshop on; 2011. p. 28–37.

[55] Dongarra JJ, Luszczek P, Petitet A. The LINPACK benchmark: Past, present, and future. Concurrency and Computation: Practice and Experience. Concurrency and Computation: Practice and Experience. 2003;15:2003.

[56] Dongarra JJ, Bunch JR, Moler CB, Stewart GW. LINPACK users' guide. vol. 8. Siam; 1979.

[57] Dongarra JJ. The LINPACK benchmark: An explanation. In: Supercomputing. Springer; 1988. p. 456–474.

[58] Huang C, Lawlor O, Kalé LV. Adaptive MPI. In: Proc. 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03); 2003. .

[59] Kamal H, Wagner A. An integrated fine-grain runtime system for MPI. Computing. 2014;96(4):293–309. Available from: http://dx.doi.org/10.1007/

s00607-013-0329-x.

[60] Kale LV, Krishnan S. CHARM++: a portable concurrent object oriented system based on C++. In: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications. OOPSLA '93. New York, NY, USA: ACM; 1993. p. 91–108. Available from: http://doi.acm.org/10.1145/165854.165874.

[61] Bosilca G, Bouteiller A, Danalis A, Herault T, Lemarinier P, Dongarra J. DAGuE: A Generic Distributed DAG Engine for High Performance Computing. Parallel Comput. 2012 January;38(1-2):37–51. Available from: http://dx.doi.org/10.1016/j.parco.2011.10.003.

[62] Nguyen T, Cicotti P, Bylaska E, Quinlan D, Baden SB. Bamboo: translating MPI applications to a latency-tolerant, data-driven form. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. SC '12; 2012. p. 39:1–39:11.

[63] Nguyen T, Baden SB. Preliminary scaling results on multiple hybrid nodes of Knights Corner and Sandy Bridge processors. In: Accepted to Third International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing; 2013. .

[64] ROSE compiler;. Available from: www.rosecompiler.org.

[65] Bylaska EJ, Weare JQ, Weare JH. Extending molecular simulation time scales: Parallel in time integrations for high-level quantum chemistry and complex force representations. The Journal of Chemical Physics. 2013;139(7):–. Available from: http://scitation.aip.org/content/aip/journal/jcp/139/7/10.1063/1.4818328.

[66] Bylaska EJ, Tsemekhman K, Baden SB, Weare JH, Jonsson H. Parallel implementation of gamma-point pseudopotential plane-wave DFT with exact exchange. Journal of Computational Chemistry. 2011;32(1):54–69.

[67] Liu QH, Jiang L, Chew WC. Large-Scale Electromagnetic Computation for Modeling and Applications [Scanning the Issue]. Proceedings of the IEEE. 2013;101(2):223–226. Available from: http://dblp.uni-trier.de/db/journals/pieee/pieee101.html#LiuJC13.

[68] Zekri AS, Sedukhin SG. The general matrix multiply-add operation on 2D torus. In: Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International; 2006. p. 8 pp.

[69] Quinn MJ. Parallel Programming in C with MPI and OpenMP. McGraw-Hill Education Group; 2003.

[70] Blackford LS, Demmel J, Dongarra J, Duff I, Hammarling S, Henry G, et al. An Updated Set of Basic Linear Algebra Subprograms (BLAS). ACM Transactions on Mathematical Software. 2001;28:135–151.

[71] Solomonik E, Demmel J. Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. EECS Department, University of California, Berkeley; 2011. UCB/EECS-2011-72. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-72.html.

[72] Marx D, Hutter J. Ab-initio Molecular Dynamics: Theory and Implementation. In: Grotendorst J, editor. Modern Methods and Algorithms of Quantum Chemistry. i ed. NIC. Forschungszentrum Jlich; 2000. p. 301–449. Publicly available at the URL: http://www2.fz-juelich.de/nic-series/Volume3/marx.pdf.

[73] Bylaska E, Tsemekhman K, Govind N, Valiev M. Large-Scale Plane-Wave-Based Density Functional Theory: Formalism, Parallelization, and Applications. In: Reimers JR, editor. Computational Methods for Large Systems: Electronic Structure Approaches for Biotechnology and Nanotechnology. John Wiley and Sons, Inc.; 2011. .

[74] Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to algorithms. 2nd ed. Cambridge, MA, USA: MIT Press; 2001.

[75] Demmel JW, Higham NJ, Schreiber R. Block LU factorization. Citeseer; 1992.

[76] Chan E, van de Geijn R, Chapman A. Managing the Complexity of Lookahead for LU Factorization with Pivoting. In: Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures. SPAA ’10. New York, NY, USA: ACM; 2010. p. 200–208. Available from: http://doi.acm.org/10.1145/1810479.1810520.

[77] Stampede User Guide;. Available from: http://www.tacc.utexas.edu/user-services/user-guides/stampede-user-guide.

[78] Alexander H, Karthikeyan V, Mikhail S, Alexander K, Roman D, Greg H, et al. Design and Implementation of the Linpack Benchmark for Single and Multi-node Systems Based on Intel Xeon Phi Coprocessor. Parallel and Distributed Processing Symposium, International. 2013;0:126–137.

[79] Kelly PHJ, Beckmann O, Field A, Baden S. Themis: Component Dependence Metadata in Adaptive Parallel Applications. Parallel Processing Letters. 2001 December;11(4):455–470.

[80] Wesseling P, Oosterlee CW. Geometric multigrid with applications to computational fluid dynamics. Journal of Computational and Applied Mathematics. 2001;128(12):311 – 334. Numerical Analysis 2000. Vol. VII: Partial Differential Equations. Available from: http://www.sciencedirect.com/science/article/pii/S0377042700005173.

[81] Bruaset AM, Tveito A. Numerical solution of partial differential equations on parallel computers. vol. 51. Springer; 2006.

[82] Feigh S, Clemens M, Weiland T. Geometric multigrid method for electro-and magnetostatic field simulations using the Conformal Finite Integration Technique. In: 2003 Copper Mountain Conference on Multigrid Methods, Copper Mountain, Colorado. Citeseer; 2003. .

[83] Nicholls A, Honig B. A rapid finite difference algorithm, utilizing successive overrelaxation to solve the Poisson–Boltzmann equation. Journal of computational chemistry. 1991;12(4):435–445.

[84] Mavriplis DJ. UNSTRUCTURED GRID TECHNIQUES. Annual Review of Fluid Mechanics. 1997;29(1):473–514.

[85] Morgan K, Peraire J. Unstructured grid finite-element methods for fluid mechanics. Reports on Progress in Physics. 1998;61(6):569. Available from: http://stacks.iop.org/0034-4885/61/i=6/a=001.

[86] Sun L, Mathur SR, Murthy JY. An Unstructured Finite-Volume Method for Incompressible Flows with Complex Immersed Boundaries. Numerical Heat Transfer, Part B: Fundamentals. 2010;58(4):217–241.

[87] Karlin I, Bhatele A, Keasler J, Chamberlain BL, Cohen J, DeVito Z, et al. Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application. In: 27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013). Boston, USA; 2013. .

[88] Karlin I, Bhatele A, Chamberlain BL, Cohen J, Devito Z, Gokhale M, et al. LULESH Programming Model and Performance Ports Overview; 2012. LLNL-TR-608824.

[89] Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory;. LLNL-TR-490254.

[90] Hirt CW, Amsden AA, Cook JL. An arbitrary Lagrangian-Eulerian computing method for all flow speeds. Journal of Computational Physics. 1974;14(3):227 – 253. Available from: http://www.sciencedirect.com/science/article/pii/0021999174900515.

[91] Benson DJ. Computational methods in Lagrangian and Eulerian hydrocodes. Computer Methods in Applied Mechanics and Engineering. 1992;99(23):235 – 394. Available from: http://www.sciencedirect.com/science/article/pii/004578259290042I.

[92] Courant R, Friedrichs K, Lewy H. On the Partial Difference Equations of Mathematical Physics. IBM J Res Dev. 1967 March;11(2):215–234. Available from: http://dx.doi.org/10.1147/rd.112.0215.

[93] Sohn A, Biswas R. Communication Studies of DMP and SMP Machines. NAS; 1997. NAS-97-004.

[94] Gupta A, Karypis G, Kumar V. Highly Scalable Parallel Algorithms for Sparse Matrix Factorization. IEEE Trans Parallel Distrib Syst. 1997;8(5):502–520.

[95] Somani AK, Sansano AM. Minimizing Overhead in Parallel Algorithms through Overlapping Communication/Computation. ICASE; 1997. 97-8.

[96] Baden SB, Fink SJ. Communication overlap in multi-tier parallel algorithms. In: Proc. of SC ’98. Orlando, Florida; 1998. .

[97] Teresco JD, Beall MW, Flaherty JE, Shephard MS. A hierarchical partition model for adaptive finite element computation. Comput Methods Appl Mech Engrg. 2000;184:269–285.

[98] Teranishi K, Raghavan P, Ng E. A new data-mapping scheme for latency-tolerant distributed sparse triangular solution. In: Supercomputing ’02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing. Los Alamitos, CA, USA: IEEE Computer Society Press; 2002. p. 1–11.

[99] Danalis A, Kim KY, Pollock L, Swany M. Transformations to Parallel Codes for Communication-Computation Overlap. In: Proceedings of the ACM/IEEE SC 2005 Conference; 2005. p. 58–68.

[100] Husbands P, Yelick K. Multithreading and One-Sided Communication in Parallel LU Factorization. In: SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing. Reno, Nevada: ACM; 2007. .

[101] Sorensen J, Baden SB. Hiding Communication Latency with Non-SPMD, Graph-Based Execution. In: Proc. 9th Intl Conf. Computational Sci. (ICCS '09). Berlin, Heidelberg: Springer-Verlag; 2009. p. 155–164.

[102] Cicotti P, Baden SB. Latency Hiding and Performance Tuning with Graph-Based Execution. In: The Seventh IEEE eScience Conference, Data-Flow Execution Models for Extreme Scale Computing (DFM 2011). Galveston Island, Texas; 2011. .

[103] Bank RE, Holst M. A New Paradigm for Parallel Adaptive Meshing Algorithms. SIAM Review. 2003;45:292–323.

[104] McCorquodale P, Colella P, Balls GT, Baden SB. Local Corrections Algorithm for Solving Poisson's Equation in Three Dimensions. Comm App Math and Comp Sci. 2007;2(1):57–81.

[105] Demmel J, Hoemmen M, Mohiyuddin M, Yelick K. Avoiding Communication in Sparse Matrix Computations. In: Proceedings of International Parallel and Distributed Processing Symposium (IPDPS); 2008. Available from: http://bebop.cs.berkeley.edu/.

[106] NVIDIA's Kepler;. Available from: http://www.nvidia.com/object/tesla-servers.html.

[107] Aji AM, Dinan J, Buntinas D, Balaji P, Feng Wc, Bisset KR, et al. MPI-ACC: An Integrated and Extensible Approach to Data Movement in Accelerator-based Systems. In: Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems. HPCC '12. Washington, DC, USA: IEEE Computer Society; 2012. p. 647–654. Available from: http://dx.doi.org/10.1109/HPCC.2012.92.

[108] Aji AM, Panwar LS, Ji F, Chabbi M, Murthy K, Balaji P, et al. On the Efficacy of GPU-integrated MPI for Scientific Applications. In: Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing. HPDC '13. New York, NY, USA: ACM; 2013. p. 191–202. Available from: http://doi.acm.org/10.1145/2462902.2462915.

[109] Wang H, Potluri S, Luo M, Singh AK, Sur S, Panda DK. MVAPICH2-GPU: Optimized GPU to GPU Communication for InfiniBand Clusters. Comput Sci. 2011 June;26(3-4):257–266. Available from: http://dx.doi.org/10.1007/s00450-011-0171-3.

[110] Potluri S, Hamidouche K, Venkatesh A, Bureddy D, Panda DK. Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs. In: Parallel Processing (ICPP), 2013 42nd International Conference on; 2013. p. 80–89.

[111] Shainer G, Ayoub A, Lui P, Liu T, Kagan M, Trott C, et al. The development of Mellanox/NVIDIA GPUDirect over InfiniBanda new model for GPU to GPU communications. Computer Science - Research and Development. 2011;26(3-4):267–273. Available from: http://dx.doi.org/10.1007/s00450-011-0157-1.

[112] Volkov V, Demmel JW. Benchmarking GPUs to Tune Dense Linear Algebra. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing. SC ’08. Piscataway, NJ, USA: IEEE Press; 2008. p. 31:1–31:11. Available from: http://dl.acm.org/citation.cfm?id=1413370.1413402.

[113] Chen Y, Cui X, Mei H. Large-scale FFT on GPU Clusters. In: Proceedings of the 24th ACM International Conference on Supercomputing. ICS ’10. New York, NY, USA: ACM; 2010. p. 315–324. Available from: http://doi.acm.org/10.1145/1810085.1810128.

[114] Dongarra J. Visit to the National University for Defense Technology Changsha, China. Oak Ridge National Laboratory; 2013. Available from: http://www.netlib.org/utk/people/JackDongarra/PAPERS/tianhe-2-dongarra-report.pdf.